STUDY AND IMPLEMENTATION OF MACHINE LEARNING ALGORITHMS OPTIMIZED FOR DISTRIBUTED MULTIDIMENSIONAL INDEXING DATABASES

TREBALL FINAL DE GRAU

March 20th, 2019

Student - Adrià Correas Grifoll
Director - Cesare Cugnasco, PhD
Tutor - Yolanda Becerra, PhD
Facultat d'Informàtica de Barcelona

Trabajo realizado para el **TFG**
Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

No se permite la reproducción total o parcial de este documento si no se cita explícitamente su procedencia.

Abstract

This project proposes optimizations for Machine Learning algorithms that benefit from distributed multidimensional indexing databases. Clustering algorithms, such as K-means, require a large amount of computational resources, and as the number of points and dimensions increase, poor scalability of the algorithms as well as the loading of large files from the disk can induce into very large execution times. A thorough analysis to study K-means scalability demonstrates poor scalability of the algorithm.

A couple of optimization implementations are presented to take advantage of the Qbeast indexing technology, which allows the use of multidimensional indexing in databases to query percentages of data over time.

It can be used to perform executions with small but representative amounts of data and improve the initialization of the algorithm. Tests have been conducted with different approaches with respect to the percentages of data consulted. Also, a performance comparison of these optimizations is shown with the standard K-means.

The tests showed that our optimizations present promising improvements compared to the standard approaches of data loading for K-means. In this way we aim high in that the work done in this project will improve and revolutionize in the near future the way in which the modern market focuses on the use of Machine Learning algorithms.

Resum

Aquest projecte proposa optimitzacions per als algorismes de Machine Learning que es beneficien de les bases de dades d'indexació multidimensionals distribuïdes. Els algorismes de clustering, com K-means, requereixen una gran quantitat de recursos computacionals, i a mesura que la quantitat de punts i dimensions augmenten, una mala escalabilitat dels algorismes a més de la lectura d'arxius grans des del disc poden induir a temps de execució molt grans. Una anàlisi exhaustiu per estudiar l'escalabilitat de K-means demostra una mala escalabilitat de l'algorisme.

Es presenten un parell d'implementacions d'optimització per aprofitar la tecnologia d'indexació Qbeast, que permet usar la indexació multidimensional en bases de dades per consultar percentatges de dades al llarg del temps.

Es pot usar per a realitzar execucions amb quantitats petites però representatives de dades i millorar la inicialització de l'algorisme. S'han realitzat proves amb diferents enfocaments pel que fa als percentatges de dades consultades. També, es mostra una comparació de rendiment d'aquestes optimitzacions amb el K-means estàndard.

Les proves van demostrar que les nostres optimitzacions presenten millores prometedores en comparació amb els enfocaments estàndard de lectura de dades per a K-means. D'aquesta manera apuntem alt en que el treball fet en aquest projecte millorarà i revolucionarà en un futur proper la manera en que el mercat modern enfoca l'ús dels algorismes de Machine Learning.

Resumen

Este proyecto propone optimizaciones para los algoritmos de Machine Learning que se benefician de las bases de datos de indexación multidimensionales distribuidas. Los algoritmos de clustering, como K-means, requieren una gran cantidad de recursos computacionales, y a medida que la cantidad de puntos y dimensiones aumentan, una mala escalabilidad de los algoritmos además de la lectura de archivos grandes desde el disco pueden inducir a tiempos de ejecución muy grandes. Un análisis exhaustivo para estudiar la escalabilidad de K-means demuestra una mala escalabilidad del algoritmo.

Se presentan un par de implementaciones de optimización para aprovechar la tecnología de indexación Qbeast, que permite usar la indexación multidimensional en bases de datos para consultar porcentajes de datos a lo largo del tiempo.

Se puede usar para realizar ejecuciones con cantidades pequeñas pero representativas de datos y mejorar la inicialización del algoritmo. Se han realizado pruebas con diferentes enfoques con respecto a los porcentajes de datos consultados. También, se muestra una comparación de rendimiento de estas optimizaciones con el K-means estándar.

Las pruebas demostraron que nuestras optimizaciones presentan mejoras prometedoras en comparación con los enfoques estándar de lectura de datos para K-means. De esta manera apuntamos alto en que el trabajo hecho en este proyecto mejorará y revolucionará en un futuro próximo la manera en que el mercado moderno enfoca el uso de los algoritmos de Machine Learning.

Index

Chapter 1

Introduction

Modern data sets are rapidly growing in size and complexity, and there is a pressing need to develop solutions to harness this wealth of data using statistical methods. Several 'next generation' data flow engines that generalize MapReduce [17] have been developed for large-scale data processing, and building machine learning functionality on these engines is a problem of great interest.

Clustering is a Machine Learning technique that involves the grouping of data points. Given a set of data points, we can use a clustering algorithm to classify each data point into a specific group. In theory, data points that are in the same group should have similar properties and/or features, while data points in different groups should have highly dissimilar properties and/or features. Take for an instance, people being diagnosed have some common symptoms and are placed in a group tagged with some label representing the disease that they suffer. Clustering is a method of unsupervised learning [13], which is able to infer patterns from a data set without reference to known or labeled outcomes, and it is a common technique for statistical data analysis used in many fields. Nowadays handling petabytes of data can be a huge overhead in many modern applications, clustering algorithms require large quantities of data and hence its computation is expensive. Clustering algorithms have been around for a long time, with continuous optimizations and different kind of approaches over the years looking to improve the state of the art according to the current needs. The Big Data era made these algorithms even more relevant with the inevitable need to classify data for daily use cases such as brain tumor detection [29], document clustering [37], face recognition [41], etc.

One of the key challenges regarding large quantities of data is the way they are handled in terms of computation of the data, the computational cost of an algorithm often makes many applications unfeasible or too expensive. A lot of times data can be represented as points in a N-Dimensional space, being the similarity between points strongly connected with proximity in the space. Having the ability to run efficient data-thinning queries, enabling the user to tackle massive data sets by analyzing smaller, but statistically relevant subsets could be a huge opportunity for a lot of modern informatic applications. In this project a new approach to handle the data in order to reduce I/O operation requirements and speed up run-time will be studied and tested for clustering algorithms.

1.1 GOAL AND PROBLEM FORMULATION

The objective of this project is to study different machine learning algorithms for clustering data, such as K-means and DBSCAN aiming to implement optimizations to achieve better handling of the data, focusing on the approach in which data is organized and comparing it against current state of the art implementations. Latency is a huge concern when managing a lot of data, storage system's I/O requirements for high amounts of data are too slow and can become a bottleneck. Precisely, we will try to improve the current scalable implementations of the clustering algorithms K-means|| [11] and DBSCAN on Spark. The optimizations that we will initially center on are: *Pre-sampling* of the data to improve run-time, use of *region* queries to avoid unneeded computation of data, *increasing data* over the iterations of the algorithm to avoid computation cost and improve run-time, and *oversampling* data. Optimizations will be selected and explained more extensively later on.

The QUAKE project at BSC is developing a prototype taking advantage of the patent pending new indexing technology Qbeast, that allows to index million of data points in a N-dimensional space efficiently and supports data analytics, machine learning and clustering tests. The focus of this work is to explore how Multidimensional indexing with efficient sampling (MIS) can improve Machine Learning algorithms and study which functionalities add to Qbeast to support it.

1.2 BACKGROUND

In this section, the basic concepts and paradigms that are needed in order to understand this project are described.

1.2.1 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key [17]. Many other models have been implemented based on this model, like Apache Spark explained later on.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed system to easily utilize the resources of a large distributed system.

1.2.2 Apache Cassandra

Apache Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers [27], while providing highly available service with no single point of failure.

Cassandra aims to run on top of an infrastructure of hundreds of nodes (possibly spread across different data centers). At this scale, small and large components fail continuously. The way Cassandra manages the persistent state in the face of these failures drives the reliability and scalability of the software systems relying on this service. While in many ways Cassandra resembles a database and shares many design and implementation strategies therewith, Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing data read efficiency.

1.2.3 Apache Spark

Apache Spark is an open-source cluster computing framework for big data processing [34]. It has emerged as the next generation big data processing engine, overtaking Hadoop MapReduce which helped ignite the big data revolution. Spark maintains MapReduce's linear scalability and fault tolerance, but extends it in a few important ways: it is much faster (100 times faster for certain applications), much easier to program in due to its rich APIs in Python, Java, Scala (and shortly R), and its core data abstraction, the distributed data frame, and it goes far beyond batch applications to support a variety of compute-intensive tasks, including interactive queries, streaming, machine learning, and graph processing.

1.2.4 Spark SQL and MLlib

Spark SQL is a module in Apache Spark that integrates relational processing with Spark's functional programming API [9]. Spark SQL lets Spark programmers leverage the benefits of relational processing (e.g., declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (e.g., machine learning). With the addition of a DataFrame API that integrates with procedural Spark code and a highly extensible optimizer, Catalyst, Spark SQL offers richer APIs and optimizations for complex need of modern data analysis while keeping the benefits of the Spark programming model.

Apache Spark is well-suited for iterative machine learning tasks. MLlib [31] provides efficient functionality for a wide range of learning settings and includes several underlying statistical. optimization, and linear algebra primitives. This platform has emerged as a widely used open-source engine. Spark is a fault-tolerant and general-purpose cluster computing system providing APIs in Java, Scala, Python, and R, along with an optimized engine that supports general execution graphs. Moreover, Spark is efficient at iterative computations and is thus well-suited for the development of large-scale machine learning applications. MLlib, Spark's distributed machine learning library, targets large-scale learning settings that benefit from data-parallelism or model-parallelism to store and operate on data or models. MLlib consists of fast and scalable implementations of standard learning algorithms for common learning settings including classification, regression, collaborative filtering, clustering, and dimensionality reduction. It also provides a variety of underlying statistics, linear algebra, and optimization primitives. The following list is compounded by the core features of MLib:

- Supported Methods and Utilities: MLlib provides fast, distributed implementations of common learning

algorithms.

- Algorithmic Optimizations: MLlib includes many optimizations to support efficient distributed learning and prediction.
- Pipeline API: Practical machine learning pipelines often involve a sequence of data preprocessing, feature extraction,model fitting and validation stages.
- Spark Integration: MLlib benefits from the various components within the Spark ecosystem.
- Documentation, Community, and Dependencies: The MLlib user guide provides extensive documentation; it describes all supported methods and utilities. It also includes several code examples along with API docs for all supported languages.

1.2.5 Clustering classification

Grouping of objects is required for various purposes in different areas of engineering, science and technology, humanities, medical science and our daily life. Take for an instance, people being diagnosed have some common symptoms and are placed in a group tagged with some label representing the disease that they suffer. The people not possessing those symptoms will not be placed in that group, and the patients grouped for that disease will be treated accordingly while patients not belonging to that group should be handled differently. Whenever we find a labeled object, we will place it into the group with the same label. However, on many occasions, no such labelling information is provided in advance and we group objects on the basis of some similarity. In generic terms, these cases are dealt under the scope of classification [18]. Precisely, the first case when the class (label) of an object is given in advance is termed as supervised classification whereas the other case when the class label is not tagged to an object in advance is termed as unsupervised classification. The main purpose behind the study of classification is to develop a tool or an algorithm, which can be used to predict the class of an unknown object, which is not labeled. This case differs from supervised classification in the manner that there is no label assigned to any pattern.

Clustering is a very essential component of various data analysis or machine learning based applications, like regression, prediction, data mining [24], etc. According to Rokach [30] clustering divides data patterns into subsets in such a way that similar patterns are clustered together. There are different clustering techniques, that have been proposed in different approaches throughout the years. Fraley and Raftery [20] suggested dividing the clustering approaches into two different groups: hierarchical and partitioning techniques. Han et al. [24] suggested the following three additional categories for applying clustering techniques: density-based, model-based and grid-based methods. In this project we focus on partitional clustering techniques, that include distance-based approaches (K-means) and density-based approaches (DBSCAN).

- K-means algorithm

The K-means clustering problem is one of the oldest and most important questions in all of computational geometry. Given an integer k and a set of n data points, the goal is to choose k centers so as to minimize the total squared distance between each point and its closest center.

K-means is a popular algorithm for clustering analysis in data mining. It aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster [39].

The K-means algorithm is a simple and fast algorithm for this problem, although it offers no approximation guarantees at all. The standard steps of K-means algorithm are:

1. Arbitrarily choose an initial k centers $C = \{c1, c2, \cdots, ck\}$.
2. For each $i \in \{1, \ldots, k\}$, set the cluster C_i to be the set of points in X that are closer to c_i than they are to c_j for all $j \neq i$.
3. For each $i \in \{1, \ldots, k\}$, set c_i to be the center of mass of all points in $C_i : c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$.
4. Repeat Steps 2 and 3 until C no longer changes

It is a standard practice to choose the initial centers uniformly at random from X. Although other approaches are currently used in the state of the art implementations that improve the cost of the algorithm.

- DBSCAN algorithm

DBSCAN is a clustering algorithm proposed by Ester et al. [19]. And it has become one of the most common clustering algorithms because it is capable of discovering arbitrary shaped clusters and eliminating noise data. The basic idea of this algorithm is finding all the core points and forming the clusters by clustering core points with all points (core or non-core) that are reachable from them. DBSCAN algorithm is based on three basic definitions: core points, directly density-reachable, and density-reachable [42]. Given a data set D, of points:

- Eps-neighborhood of a point p is the neighborhood of p ∈ D within a radius eps.
- Definition 1: A point p is a core point if it has neighbors within a given radius (eps), and the number of neighbors is at least minpts (which is a threshold). In this case, the number of neighbors is called density.
- Definition 2: A point y is directly density-reachable from x if y is within eps-neighborhood of x and x is a core point.
- Definition 3: A point y is density-reachable from x if there is a chain of points $p_1, p_2, \ldots, p_n$, with $p_1 = x, p_n = y$ and $p_i + 1$ is directly density-reachable from p_i for all 1 <= i < n, $p_i \in D$.

The algorithm starts with an arbitrary point p in D and checks its eps-neighborhood. If the eps-neighborhood size is bigger than pre-defined number minpts, the code generates a new cluster C. It then retrieves all density reachable points from p in D, and adds them to the cluster C. Otherwise, if the eps-neighborhood contains less than minpts points, then p is marked as noise. The computational complexity of the algorithm is $O(n^2)$ where n is the number of data points. If spatial indexing [14] were to be used, the complexity would be reduced to $O(n * log(n))$.

1.2.6 Qbeast

Qbeast technology has been developed at BSC-CNS, this technology combines Multidimensional Indexing with efficient data Sampling (MIS) that allows to improve the approach in which scientific data is analyzed.

It provides a novel indexing algorithm, designed to be implemented on top of modern distributed key-value databases, like Cassandra. Its design employs de-normalization techniques to increase the scalability and the parallelism of multidimensional queries. It also allows to run efficient data-thinning queries, enabling the user to tackle massive data sets by analyzing smaller, but statistically relevant, subsets.

The classical indexing algorithms, such as R-trees[22], were designed to take advantage of the hardware's characteristics at the time. At the time of its design the seek latency of the hard disks was the biggest performance bottleneck, RAM was extremely expensive, and databases ran on monolithic single core machines. The architecture of the computers has changed dramatically since then. The CPU's clock frequency growth has stopped, giving the way to new architectures built on multiple cores. Modern CPUs have also adopted a Non-Uniform Memory Access (NUMA) design, which distinguishes between the local and the remote RAM, opening new possibilities for performance improvement when applications embrace share-nothing parallelism.

The structure that Qbeast used in a previous version of the framework is called D8-tree[15], refered as Denormalized Octa-tree. The characteristic of the D8-tree is to replicate the elements with higher priority on the top levels of the tree so that the data contained within each level is also stored in the lower ones. Figure 1.1 shows a representation of the first and second levels of a D8-tree where each node can accommodate maximum one element. It is possible to see how the elements available at the first level, the red cubes, are also accessible at the second level of the tree. Figure 1.2 shows the structure of an Octa-tree where denser zones are more partitioned thus making data thinning more complex.

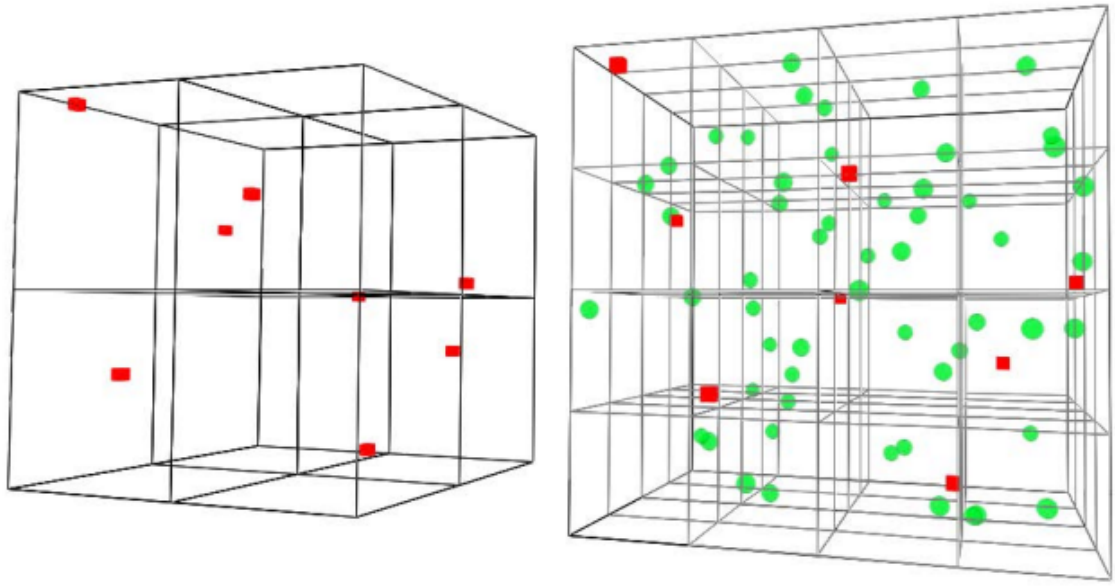


Figure 1.1: The first(left) and second level(right) of a D8-tree

Indeed, if we want to sample one point from the top-left quadrant, the one with the green pyramids and blue spheres, we should access each of the 15 partitions reading all the elements and then randomly select the one we need in memory. With the D8-tree, instead, we would need to read just one point from the first level of the tree as long as it is already a random sample of the underlying space area.

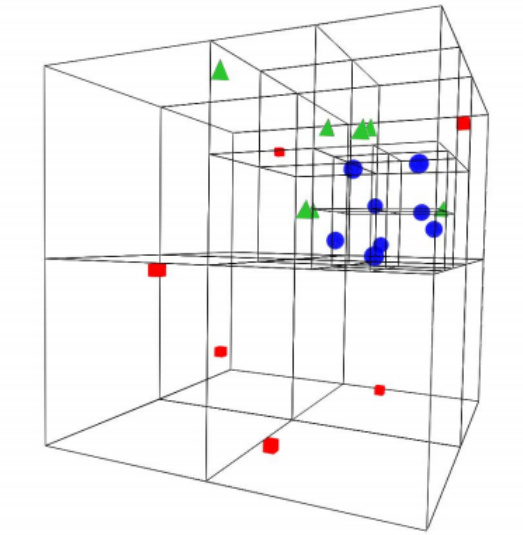


Figure 1.2: Octa-tree structure

A second implementation called AOTree (Asymptotic Outlook Tree), which is the one currently used, puts its weight in how to write data, apart from maintaining the read optimization that its ancestor has. This reduces

the requirements of space in the disk without compromising read times.

The idea of a "max-height" global for the index is replaced for a maximum capacity in each cube. Then, the cube will only replicate the data needed to perform a regular jump.

This technology can be harnessed into many applications. On one side, scientific simulations generate complex data sets where multiple characteristics describe each item. For instance, a particle might have a space position (x, y, z) at a given time (t). If we want to find all elements within the same area and period, we either have to scan the whole data set, or we must organize the data so that all items in the same space and time are stored together, which is very complex. Using Multidimensional Indexing (MI) to cluster and to organize similar data together solves this issue. On the other side, we can relax the precision of the results to speed up analytical workloads within a specific interval of confidence.

Now focusing on clustering algorithms, these kind of algorithms usually require to load and compute a very high amount of data, the access to disk can be slow and become a significant bottleneck. Being able to load a percentage of the data very fast and start the algorithm with a sample of the whole data can be very positive for these algorithms. Instead of waiting for the whole data to be loaded before starting computation of the algorithm it could be possible to start iterations and load data "on the go".

1.3 STATE OF THE ART

In this section, a revision of two key machine learning clustering algorithms state of the art implementations is presented.

1.3.1 K-means++

The idea of K-means algorithm goes back to Hugo Steinhaus in 1957 [39], since then a lot of implementations have been proposed to improve the quality and run-time of the algorithm. Big Data has been a hot topic for several years and as technology evolves new ideas and optimizations try to improve the state of the art.

MLlib from Apache Spark has a current implementation of K-means [10], that improves the initialization process using an optimization that outperforms the standard random initialization. MLlib also provides a scalable version of the algorithm K-means++, called K-means|| [11] that allows to run the algorithm in parallel to increase the speed of the algorithm. Moreover, MLlib includes a Streaming K-means and Bisecting K-means [37] variant.

1.3.2 Scalable DBSCAN

Unlike K-means, DBSCAN is not currently supported in Apache Spark, as there isn't an implementation in MLlib. Though, there are some public unofficial implementations on top of Apache Spark trying to take advantage of the framework, unfortunately benchmarks show unpromising results. Han et al. [23] defines how an efficient

DBSCAN algorithm on top of Apache Spark could be implemented, but its implementation is not public. ELKI [33] is an open-source library for data analysis that includes an implementation of DBSCAN (as well as K-means), although it can only be executed in a single node because it doesn't support parallel execution, it is widely known and used for current data analysis projects, (in some cases even outperforming parallel executions on top of Apache Spark). There are a lot of variants of DBSCAN, such as OPTICS that successfully solves some of the issues of the algorithm, the focus in the project consists in how to handle the data so that the algorithm can take advantage of that, for this reason we will ignore some improvements and optimizations of the algorithm that focus on improving the core drawbacks of DBSCAN. Moreover, HPDBSCAN is a parallel implementation of DBSCAN using OpenMP and openmpi to parallelize the algorithm and it is proved to scale very well when set up correctly in a cluster. The following tables show benchmarks [32] from different implementations of DBSCAN:

	Data points	HDF5 size	CSV size	SSV with Ids
Twitter Small	3.704.351	57 MB	67 MB	88 MB
Twitter Big	16.602.137	254 MB	289 MB	390 MB

Implementation	**Repository**	**Version date**
HPDBSCAN	bitbucket.org/markus.goetz/hpdbscan	10/09/2015
Spark DBSCAN	github.com/alitouka/spark_dbscan	22/02/2015
RDD DBSCAN	github.com/irvingc/dbscan-on-spark	14/06/2016
DBSCAN on Spark	github.com/mraad/dbscan-spark	30/01/2016

Number of cores	1	2	4	8	16	32
HPDBSCAN MPI	114	59	30	16	8	6
PDSDBSCAN MPI	288	162	106	90	85	88
ELKI	997	-	-	-	-	-
RDD-DBSCAN	7311	3521	1994	1219	889	832
DBSCAN on Spark	1105	574	330	174	150	147

Table 1.1: Run-times in seconds of the different implementations

The previous table 1.1 shows run-time measurements on Small data set, using a few number of cores. HPDBSCAN (C++ implementation) on MPI only mode performs best in all cases and scales well. Second in terms of run-time is C++ PDSDBSCAN (MPI variant), however, the scalability beyond 8 cores is already limited. Even the Java ELKI which is optimised for a serial execution is much slower than the C++ implementations. The implementations for Spark are even slower (for all of them, 912 initial input partitions were used). When running on many cores, the Spark implementations beat ELKI but they are still by one (DBSCAN on Spark) or two (RDD-DBSCAN) magnitudes slower than HPDBSCAN and do not scale as well. Further analysis and benchmarks are tested thouroughly in [32].

1.4 SCOPE

The first task is to make a large search of different clustering algorithms to understand them and analyze if we can take advantage of our approach of handling the data to optimize them.

Our focus will be to analyze what features can be included in Qbeast to improve ML algorithms.

After a precise analysis of the implementations of the algorithms and tests a decision will be made upon selecting which algorithms are chosen for a thorough study and possible optimizations. Then, we will validate the veracity of the solution of our optimizations.

Finally, we will benchmark the algorithms, including strong scaling and weak scaling tests. For that we will need to set up a fair Spark cluster environment at Marenostrum4.

1.5 STAKEHOLDERS

- **Developer**: This is the person in charge of research, document, benchmark and implement all the required tasks. In addition, he is responsible for the project management and the writing of the report and all the required documentation. This actor works as agreed with the director and he is, ultimately, the person in charge to accomplish the deadlines.
- **Director**: The director is the main responsible for guiding, giving advice and, in general, helping the developer. His action is key to determine possible errors in the project, both in its proposal and execution. In particular, Cesare Cugnasco, Research Engineer at BSC-CNS, has acted as director.
- **Beneficiaries**: Many businesses are interested in big data applications since we still lack on efficiency in terms of understanding tons of data and the ability to extract valuable information from it. Optimizing clustering algorithms would help creating better applications for real modern big data challenges and hence, would benefit both users and enterprises. In particular, at BSC-CNS some projects will be directly benefited: the european project IBiDaaS, QUAKE project for data analytics and collaborations with other companies like LaCaixa, FCA, Telefonica, etc.

1.6 METHODOLOGY

A set of benchmarks for current state of the art implementations will be made in Minerva cluster(DAC), Marenostrum(BSC) and AWS from Amazon, to have in comparaison for our future tests. Official datasets for benchmarking will be used [32].

Our implementations will be built with Scala on top of Apache Spark framework, and the data will be indexed in Cassandra using Qbeast technology for indexing. The algorithm will query the data from Cassandra using a cassandra connector for Apache Spark. The optimizations will be centered in taking advantage of the methodology in which Qbeast indexes data, when space is partitioned for scalable versions of clustering

algorithms we could query the data spatially and easily from Cassandra. Qbeast offers some parameters like density that can be useful for decisions like selecting initial k cluster for K-means or how to merge two different clusters in DBSCAN (partitioned).

Some other optimizations will include pre-sampling of the data, using a percentage of the total data for selecting initial clusters for K-means may improve significantly a run of the algorithm with the total data and custom centers set from the pre-sampled previous run of the algorithm. Also, incrementally increase the data as the algorithm is being executed could show interesting results.

Furthermore, we will evaluate the optimization's improvements in terms of run-time, computation cost and quality of the output. For instance, we will measure the speedup obtained when running K-means with custom centers from a pre-sampled execution and compare it against a normal execution.

1.7 VALIDATION METHODS

We will use a hypothesis, test and validation approach during our development, along with the combination of weekly meetings with the project director and tutor for supervision will ensure the validation the objectives and the correct evolution of the project.

1.8 POSSIBLE OBSTACLES AND SOLUTIONS

The higher risk is that theoretical advantages could result in an insufficient considerable performance improvement. If that were the case, we would be aware of that thanks to this project. The path followed when a new approach is proposed to solve an issue regarding state of the art implementations usually means obstacles will appear, and sometimes unexpectedly. As it has happened before [32], out of memory exceptions are a common issue when running Spark implementations. Even if a node has a lot of memory, the memory is shared between cores in the node. Hence, the number of cores used on each node has to be restricted using the -executor-cores parameter, thus reducing parallelism, but leaving each thread more RAM. An issue that may be hard to overcome is the strong dependency between partitioned clusters for scalable K-means++, even if every partition can be computed efficiently between nodes there will always be a dependency in communicating each of the partitioned cluster centers between them. Moreover, pre-sampled data percentage can be different for each data set so that must be taken into account. Also, The cost of indexing the data in Cassandra could become a bottleneck, if there is too much data to be indexed "fast" a possible idea could be indexing incrementally or in samples. In addition, scalable implementations of DBSCAN require a complex process to merge different clusters from different partitions, doing this process efficiently and with good results could be the most important challenge for a scalable DBSCAN.

Chapter 2

Project planning

In this section, a description of the project phases and the resources and requirements for each one is provided along with the planning and scheduling.

The expected project duration is of about 4 months. The project started on 2nd February, 2019 and the estimated end is on 25th June, 2019, before the presentation sessions begin.

Initially we thought about optimizing both K-means and DBSCAN following the scheduling of the different tasks described next. Along with discarding DBSCAN optimizations, some plan deviations will be pointed out at the end of this chapter.

2.1 TASK DESCRIPTION

2.1.1 Background in distributed computing

In the last months I have been studying distributed computing as a requirement for my job position at BSC, and as a preparation for my TFG. The study mainly consisted in firstly, getting to know the general concepts to then, start reading papers about MapReduce [16] [17] and Apache Spark [8] [34]. From there I started to get familiar with scala, which is a programming language that has good cohesion with distributed computing and, in particular, Apache Spark framework. Moreover, I started to take a look at MLlib [31] for the particular application that this project wants to examine. This process took several months because I was at the same time studying the lectures for a semester at FIB. This task did not require any material resources besides printed documents, but it did require human resources and time to read and understand all the information.

2.1.2 Background in NoSQL databases

NoSQL databases was another topic that I had to get to know for the first time, understanding how distributed databases work was required because in this project we will be using the indexing algorithm (Qbeast) for the NoSQL database Apache Cassandra. For that, research [27] was made in order to understand how NoSQL databases work. Furthermore, the cluster Minerva at DAC was used to run Cassandra in different nodes for testing. This task only required human resources in the research phase. On the other hand, hardware and software resources were used to setup a Cassandra instance in Minerva cluster at the last phase.

2.1.3 Background in clustering algorithms

There are plenty of clustering algorithms, and each of them have different implementations for sequential and scalable techniques. This task is focused on understanding the state of the art of the current clustering algorithms and in particular, focus on the scalable versions of the algorithms. Approximately one month was required in order to get the sufficient background about clustering algorithms, and after that a precise analysis was made to select which algorithms we are going to center on. K-means and DBSCAN will be the main algorithms in which we will test possible optimizations. This study only took human resources besides the print of documents.

2.1.4 Analyze and implement possible optimizations

This task consists in a thorough analysis of the algorithms K-means and DBSCAN, to then study which optimizations to test for each algorithm. The optimizations will be centered on these techniques:

- **Pre-sampling** the data before running the algorithm. For instance, in K-means we can take advantage of this technique to select initial centers, running the algorithm with a small percentage of all the data can be very fast, and using the final centers of that run as the initial centers for the run with all the data could be beneficial.
- **Increasing the data**, with this technique we want to run the algorithm with an small percentage of data, and increase that percentage of data over the iterations of the algorithm.
- **Region Query**, some of the density-based algorithms like DBSCAN can take advantage of a partition of the space for the scalable optimizations, where each partition is executed in an independent node, doing region queries to select the data directly in a region instead of computing all the data and then selecting a region could be beneficial.
- **Oversampling of data**, when clustering data, some regions of data are dense enough that may not need to be computed to know they form a cluster, Qbeast can provide the density of a region easily. If we could avoid computing some regions of data because they are dense enough, we could improve the run-time.

Also, this task includes the implementation of such optimizations, hence this task will require a considerable large amount of time.

As a disclaimer, since the project is still at an early stage ideas for optimizations and experiments may differ from the actual ones that may be performed.

The resources needed for this task are:

1. **Human Resources**
    - The developer needs to analyze and implement.
2. **Hardware Resources**
    - Personal laptop used in all tasks for this project.
    - Minerva cluster at DAC to do tests in a real cluster.
3. **Software Resources**
    - OpenSuse 15 OS used in all the tasks.
    - Scala: used to build the scripts for the experiments.
    - Libraries: Set of libraries and dependencies to run the experiments, such as MLlib, spark cassandra connector, etc.

2.1.5 Setup benchmark environment

Before doing any test, we need to define the setup where we will be performing the experiments. On one hand, we will have two kinds of data sets, a synthetic dataset created by us from 1M to 3M 3D/4D points and a real data set. The real data was obtained by Junjun Yin from the National Center for Supercomputing Application (NCSA) using the Twitter streaming API. This data set contains 3 704 351 longitude/latitude points and is available at the scientific data storage and sharing platform B2SHARE. There, the data is contained in file twitterSmall.h5.h5. A bigger Twitter data set twitter.h5.h5 from the same B2SHARE location covers whole of June 2014 containing of 16 602 137 data points.

On the other hand, the experiments will be performed in different clusters. We intend to perform the experiments in Minerva cluster (at DAC), Marenostrum (at BSC), and AWS.

The resources needed for this task are:

1. **Human Resources**
    - The developer needs to setup the benchmark environment by installing all libraries and software required to make experiments.
    - Help from the BSC's support team.
2. **Hardware Resources**
    - Personal laptop used in all tasks for this project.

- Minerva cluster at DAC where tests will be executed.
- Marenostrum cluster at BSC where tests will be executed.
- AWS cluster from Amazon where tests will be executed.

3. **Software Resources**
   - OpenSuse 15 OS used in all the tasks.
   - Scala: used to implement the optimizations.
   - Python: used to implement scripts and tests.
   - Libraries: Set of libraries and dependencies to run the experiments, such as MLlib, spark cassandra connector, etc.

2.1.6 Perform the experiments

During this task, a set of benchmark experiments will be made following a test and validation approach method. Once we have analyzed properly all the possible optimizations, we will try to test them in an order of achievable-approach manner to validate them, thus the optimizations that we think are easier to implement and more likely to success will be made first. Each experiment will be executed several times and measure of the mean of each run-time will be made as a standard method. In Minerva, we will make sure all caches are dropped after each execution to avoid cache exploitation for concurrent executions. The same executions will be made in different clusters to be able to compare how the scalability is affected in different environments.

The resources needed for this task are:

1. **Human Resources**
   - Setup and run correctly the experiments
2. **Hardware Resources**
   - Personal laptop used in all tasks for this project.
   - Minerva cluster at DAC where the experiments will be performed.
   - Marenostrum cluster at BSC where the experiments will be performed.
3. **Software Resources**
   - OpenSuse 15 OS used in all the tasks.
   - Scala: used to build the scripts for the experiments.

2.2 ESTIMATED TIME

In Table 2.1 the estimated time (in hours) needed to accomplish each task is shown.

Task	Estimated duration time (h)
Background in distributed computing	55
Background in NoSQL databases	60
Background in clustering algorithms	70
Analyze possible optimizations	105
Setup benchmark environment	50
Perform the experiments	80
Final stage	30
Total	450

Table 2.1: Estimation of the duration time (hours) of the tasks

2.3 GANTT CHART

Figure 2.1 (next page) shows the tasks planned for the project in a Gantt chart.

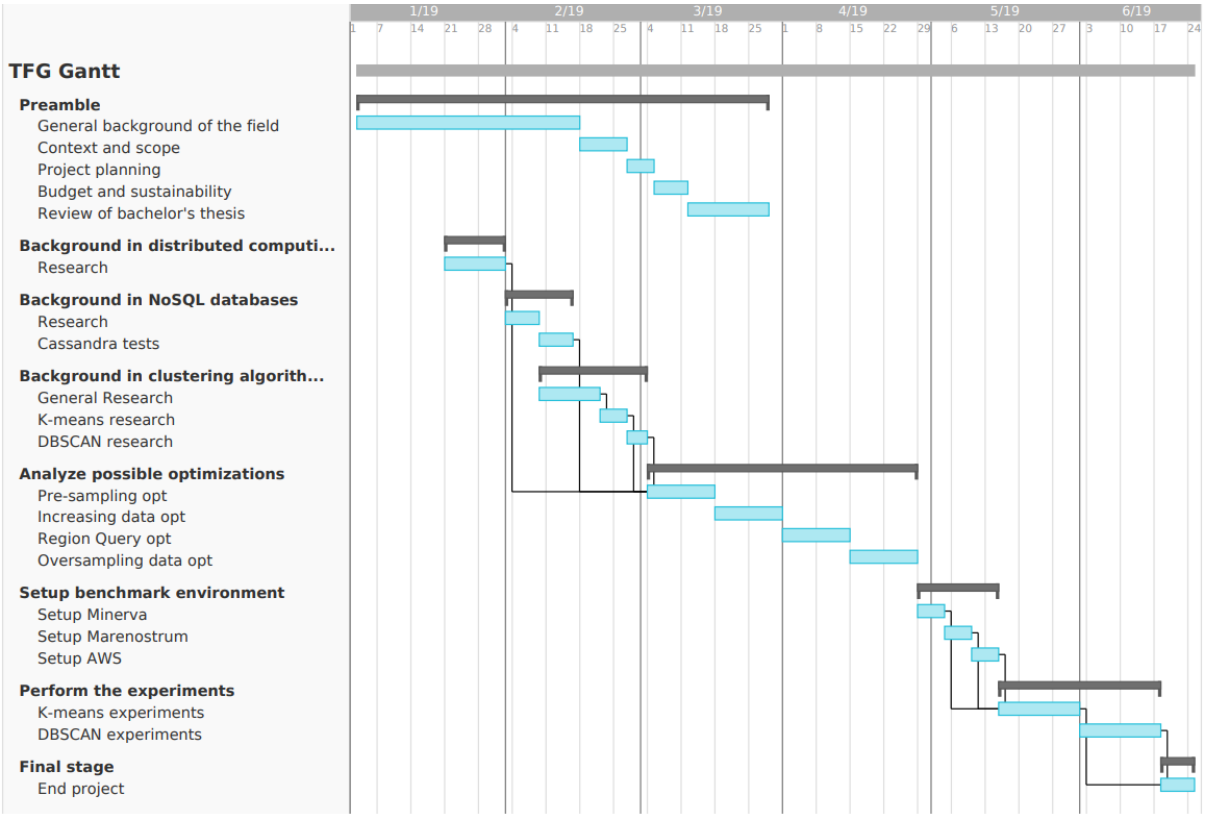


Figure 2.1: Gantt chart of the project

2.4 ALTERNATIVES AND ACTION PLAN

As it has been explained, the use of a test and validations lock methodology (agile) will allow to revise and adapt dynamically this initial planning. Although the main ideas for optimizations and duration time are set, if alterations occur, such as bugs or a task has a different duration from the expected one, the planning will be modified accordingly. If a task lasts more than expected, a later task will have to be shortened or, in the worst case cut out of the action plan. On the other hand, if a task has less duration than the expected, the next task will start as soon as possible.

As pointed out previously, the optimizations are decreasingly ordered based on the estimated gain in terms of executions time and feasible implementation. Weekly meetings have been arranged with the project director and tutor to detect possible deviations from the original planning with enough time to be able to correct them.

Even though we will try to prevent possible deviations, if we were too much behind schedule an alternative solution would be to cut out less important tests or reduce its number for each optimization, being able to present all the results but in this case with less quantity or accuracy. Moreover, coming across with a bug will mean:

- Firstly, if the developer isn't able to fix it by himself, he will ask for help to the tutor/director of the project.
- If we cannot overcome a bug we would decide whether to rollback to a previous version or change the code (and possibly the end solution) to avoid part of the code related to the bug.

Summarising, with an estimated dedication of 30 hours per week and a total budget of (16x30) 480 hours and the alternatives mentioned above, the project planning is feasible.

2.5 PLAN DEVIATIONS

2.5.1 Goal and contextualization

The goal of the project was to study how Machine Learning algorithms can take advantage of multidimensional indexing with efficient sampling (MIS).

We optimized K-means algorithm, which is a clustering algorithm that is used to classify data in an unsupervised way. We want to integrate the new technology Qbeast to handle the data for the algorithm, it will work as a multidimensional indexing algorithm on top of Apache Cassandra. Moreover, we are using Apache Spark because it is a state of the art framework for distributed computing, which supports a K-means library and a Cassandra connector, that we will use to query the data from the NoSQL database.

K-means is one of the most famous clustering algorithms that is widely used in the market, use-cases go from brain tumor detection in the eHealth field to document classification. Even though it is a common algorithm, it has some problems because it requires a lot of computation and its scalability is bad.

We decided to use Scala programming language to implement the optimizations because it's very popular in Big Data projects and supports the technologies that we want to use. We can import all the spark libraries which contain K-means algorithm very easily, as well as using the spark connector for Cassandra. The reason why we did not choose Python, which also supports ML algorithms, is that it does not provide distributed ML libraries. Also, the most mature and popular distributed Machine Learning is written in Scala.

Though, to validate the solution of our optimizations we used silhouette score which is an sklearn algorithm for Python.

2.5.2 Project plan deviations

Initially we thought about implementing and testing DBSCAN algorithm. But, as research projects usually suffer plan deviations, some changes have happened regarding the initial planification. The optimizations for DBSCAN algorithm have been discarded of the project because the complexity to implement some scalable parts of the algorithm using Spark(Spark MLlib doesn't support DBSCAN) was too costly along with the time to implement and test K-means optimizations which was already enough workload. So, benchmark tests for DBSCAN have also been discarded.

Moreover, a point that we didn't take into account in the initial planification is that a lot of time was needed to setup the cluster environment at Marenostrum, analyze the scalability of the algorithms and understand that we are doing it right, so that the tests that we are gonna perform make sense in comparison between them.

For these reasons, we have re-defined the optimizations related to K-means, reduced to a couple of optimizations (Bi-phase and Multi-phases) that were the most promising to show good results. The main objective to show that the K-means algorithm by itself doesn't scale well and that with our optimizations we can obtain significant Speedups remained.

Furthermore, initially we thought to use two different data sets from Twitter that are publicly available, but we saw that we needed higher amounts of data as well as more data sets to perform weak scaling tests. That is why we ended up using a single large data set from Open Street Map where we took the Geo-localization parameters and skewed the whole data set to form different data set sizes. Mainly we ended up using a small skewed data set (2,6G) and a bigger one of 26G.

Also, with the optimizations that we chose for K-means we selected which features need to be implemented for Qbeast to support ML and our optimizations. Therefore, the last section about the Qbeast results will be focused on comparing the results with an approximation of how efficient sampling will work.

The overall cost of the project has not changed, the developer has worked the amount of hours that was initially planned. Minerva cluster was mainly used to test the algorithms correctness before performing tests in a high-end cluster like Marenostrum4. We also thought to perform tests in AmazonWebServices clusters, but we finally discarded it because the tests performed at Marenostrum4 were already conclusive.

Finally, the state of the project plan is almost close to the end, further tests need to be performed and finally write down everything in the memory.

2.5.3 Methodology

There haven't been significant methodology deviations. Though, a methodology that we had not mentioned before and is worth to be pointed out is the performance metrics that we chose to understand the scalability of the algorithms, before we went on to start performing the experiments.

2.5.4 Analysis and alternatives

There is a lack for specialized frameworks for multidimensional data. We chose Qbeast on top of Cassandra as the indexing algorithm because it shows better performance than native Cassandra, that does not provide spatial sampling and efficient sampling(percentages of data). It also shows better performance than multidimensional PostgreSQL.

2.5.5 Laws and regulations identification

We are using technologies from Apache Software Foundation. Since Spark and Cassandra are open-source they are freely available to the public. In this case, the license to which refers is one of the most known: Apache License 2.0 [7]. To sum up, it has some conditions and limitations:

- Allows its commercial use, distribution, modification, patented use and private use.
- The license must remain and is susceptible to state changes.
- Limits the responsability, use of brand and warranty.

On the other hand, Qbeast is a brand new technology, patent pending protected IP that will be published in April 2020.

2.6 BUDGET AND SUSTAINABILITY

2.6.1 Sustainability self-assessment

The poll is about the self-evaluation of sustainability knowledge. Its main goal is to get information of the knowledge and competences in sustainability from teachers and students in the Spanish university system.

The poll is divided in three dimensions: social dimension, environmental dimension and economical dimension. A set of question are targeting each dimension individually.

My personal opinion about the knowledge acquired about sustainability in the university system is that it should be learnt exclusively in a compulsory subject. I was lucky to course APC which helped me understand better some of the concepts about sustainability, but most of my colleagues did not take that course and hence,

in my opinion they have not learnt enough concepts about sustainability during the degree. One way in which the university could include the variable of sustainability more effectively could be integrating that in a project. I have done many projects during the degree and despite TFG, none includes concepts about sustainability.

We are aware of the issues that pollution provokes, we know that we should be as sustainable as possible. With digitization, sometimes we forget that small things like sending an email or doing a google search means CO2 emissions to the atmosphere. And if we scale it to millions of people and hundreds of services we can realise that we can send tons of CO2 with small actions.

I think that digitization is unstoppable and technology is our ally, but we need to be really aware of the consequences that it produces, and we need to think always with the sustainability variable at our side.

But for that, it is really important that more awareness is given in the education sector, which is probably the most important one since in our case, engineers can have a high impact when deciding how to implement projects to be respectful towards the environment.

2.6.2 Project budget

In order to carry out this project, a set of Hardware and Software resources mentioned earlier will be used, next we will detail the budget of these two resources along with Human resources as they are also part of the total budget. In addition, risk and indirect costs will be taken into consideration and are described in this section.

2.6.2.1 Human resources budget

This project will be performed by a single person. This person will need to act for all the basic roles in a project, Hence, this person will need to be a project manager, software developer and, since we need to perform benchmark tests in this project this person will also act as a tester. Each role will be responsible for different tasks distributed in the total of 450 hours. In table 2.2 an estimation of the human resources budget is provided.

Tasks	**Role**	**#Hours**	**€/hour**	**Total cost (€)**
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16	Software developer	260	35	9.100
11, 12, 13, 14, 15	Tester	150	30	4.500
7, 8, 9, 10, 16	Project manager	40	50	2.000
	Total	450		15.600

Table 2.2: Human resources budget(€) and task association per role

Each number in column Tasks represents a different task from the Gantt chart provided in the project planning section. Number 1 would be the first task counting from: *Research* - Background in distributed computing. Number 16 would be the last task from the chart: *End project* - Final stage.

2.6.2.2 Hardware budget

A set of hardware resources will be needed in order to implement the optimizations previously explained and to perform all the needed benchmarks. Table 2.3 shows the estimation cost of the hardware required for this project taking into account the cost per unit, useful life and amortization of each product.

Product	**Units**	**Cost/u (€)**	**Useful life (years)**	**Amortization (€)**
Dell Latitude 7480	1	1.567,4	5	130,6
Minerva	1	0	-	0
Marenostrum	1	-	-	0
Total		1.567,4		130,6

Table 2.3: Hardware budget

In relation to the amortization of the products, the amortization cost for the Dell Latitude 7480 is based taking into account 5 months of its use. Moreover, Minerva is a 6 year old cluster, and we consider that it has already been amortized. As for Marenostrum costs, we will minimize the use of it, most of the implementations and analysis will be performed in Minerva, thus, we will consider this costs as negligible. Finally, the cost of AWS t3.small virtual machine is 0.0208 €/h and we will use it for two weeks (60 h).

2.6.2.3 Software budget

Many Software resources will be need in order to carry out the project. All the software that will be used for the project is free since the university email and business partnerships allow us to get plenty of software for free. Although some of the software is already free without the need of these tools, some software resources restrict their content for the standard version, but we can unlock the education/professional version thanks to the emails or business partnerships.

Product	**Units**	**Cost/u (€)**	**Useful life (years)**	**Amortization**
Overleaf	1	0	-	0
Git Hub	1	0	-	0
Google Drive	1	0	-	0
Mendeley	1	55	-	0
IntelliJ Professional	1	499	-	0
Teamgantt	1	0	-	0
OpenSUSE Leap 15	1	0	-	0
ParaView	1	0	-	0
Vim	1	0	-	0
Zoho Sprints	1	120	-	0
Total		0		0

Table 2.4: Software budget

Table 2.4 shows the estimation cost of the software required for this project taking into account the cost per unit, useful life and amortization of each product.

2.6.2.4 Risk costs

In addition, an estimation of unexpected costs is shown in table 2.5, derived from the potential deviation of the project plan.

Role	**Hours**	**€/hour**	**Total cost (€)**
Software developer	25	35	875
Tester	25	30	750
Project manager	10	50	500
Total	60		2.125

Table 2.5: Risk costs

We consider the potential incidents listed next:

- Delay in any task.
- Issues regarding the clusters where we will be performing the tests.
- Issues regarding the implementation of any optimization.

In order to resolve the potential incidents the plan is to either work more hours to end in time or more hours per day to comply the schedule.

2.6.2.5 Indirect costs

Indirect costs when carrying out the project would be transport, electricity and connection to the internet. In table 2.6 a list of the indirect costs is shown.

Element	**Cost (€)**	**Units**	**Total cost (€)**
Transport (T-jove)	105	2	210
Internet	35/month	4 months	140
Electricity	0,135/kWh	2000 kWh	270
Total			620

Table 2.6: Estimation of the indirect costs for the project

2.6.2.6 Total budget

Finally, to measure the total budget of the project shown in table 2.7, we add all the budgets provided above including 4% of contingency that has been added over the total cost in order to cover unexpected costs.

Concept	**Total cost (€)**
Human resources budget	15.600
Hardware budget	131
Software budget	0
Risk costs	2.125
Indirect costs	620
Total partial	18.476
Contingency (4%)	739
Total	19.215

Table 2.7: Total budget

2.6.3 Budget control

For the project we will monitor the budget in order to control it in case any deviation occurs during the project's development, after the end of each task we will check whether the budget should be updated according to the duration of that task and the remaining hours left for the next tasks. We will detect any possible deviation due to variance in cost or consumption (and time) since we track both the amount of hours per task and the cost (resources) spent in each task.

Moreover, all of this is done taking into account that we already have a contingency plan in terms of both budget and time.

2.6.4 Sustainability report

In this section we analyse the sustainability of the project in three different dimensions based on the sustainability matrix shown in table 2.8.

	PPP	**Useful life**	**Risks**
Environmental	8/10	8/10	-1/-10
Economical	7/10	6/10	-2/-10
Social	6/10	8/10	-1/-10

Table 2.8: Sustainability matrix

2.6.4.1 Environmental dimension

The expected consumption in the development stage of the project is the consumption of the electricity in the work environment, both for the basic equipment like the laptop and any cluster used for the tests. To optimize time and energy when doing the tests, we want to run multiple executions per test to take advantage of the data from Cassandra already cached, instead of doing a single execution per test. As for the useful life, the goal of the project is to optimize clustering algorithms, and by doing that, we would be reducing the energy consumption

of the algorithms (in run-time). Thus, this project could potentially improve the ecological footprint of current solutions. Furthermore, the current proposed solution would avoid computation time thanks to range queries and pre-sampling techniques.

As for the reduction of the environmental impact, next are listed some methods that we will try to carry out:

- Working from home to avoid the transport cost to university.
- When performing some of the tests multiple executions must be carried out, we will perform those executions sequentially to take advantage of the resource allocation from the initial job, instead of submitting a job for each execution.
- Any given code implementation already made that we can use will be taken into account instead of losing time and effort to implement it ourselves, that would ultimately induce into an environmental cost.

2.6.4.2 Economical dimension

As pointed out previously, the quantification of all the costs has been shown for all types of resources. The proposed solution has been concrete for our project, but it is hard to compare it with other implementations. We can observe that the budget can easily increase over time, we hope that the work in this project will be worth for future applications in which could contribute. Since our solution will be an optimization in run-time and resource consumption it will also be beneficial in the economical dimension.

2.6.4.3 Social dimension

Personally, this project will bring me the necessary experience for the labour market, which is not acquired during the rest of the degree. From the point of view of environment and professional relations, towards meetings to discuss the direction of the project. Also the fact to work against deadlines and the responsibility of the mistakes that it produces.

Being Big Data such a hot topic and having the possibility to improve the current state of the art of data classification is a huge motivational factor.

Although it can be left unnoticed for a lot of people, mainly because applications in data analysis are usually hidden behind products and applications, a wide variety of current real use-cases such as medical field, air traveling field, etc, can be benefited considerably if these use-cases are able to classify their data in a better way.

Chapter 3

Selected optimizations implementation

The goal of the experiment is to test and compare how our optimizations improve the state of the art of K-means in terms of run-time and computation cost. Our optimizations aim to take advantage of Multidimensional Indexing with efficient Sampling (MIS), that allows us to obtain samples of data much faster than classic management of data obtained from disks. These optimizations will allow to obtain K-means convergence much faster and hence, improve many K-means use-cases in the modern world. For instance, machines will be our future doctors, given the symptoms of the patient they will be able to detect the illness of the patient before you leave the room. Moreover, any given application with a process of clustering which can slow down many product pipelines will be able to improve and get a solution for their product in a faster way.

3.1 Optimizations

The optimizations have been implemented in Scala using Apache Spark framework to distribute the computation in a cluster. We have implemented each optimization with two kinds of data source, one reading data from disk to first analyze the viability of the optimizations and the other one reading from Cassandra on top of Qbeast indexation.

3.1.1 Bi-phase

This optimization consists of two main phases, the first phase takes care of a K-means execution with a small percentage of all data. After that, another K-means execution is performed, but now with all the data and centers initialized from the output of the previous run. K-means implementation from MLlib allows us to set custom centers before starting the algorithm, hence we can get the output from one execution, and use it to initialize a different one. In figure 3.1 we can observe a representation of the data along with a formal notation right after.

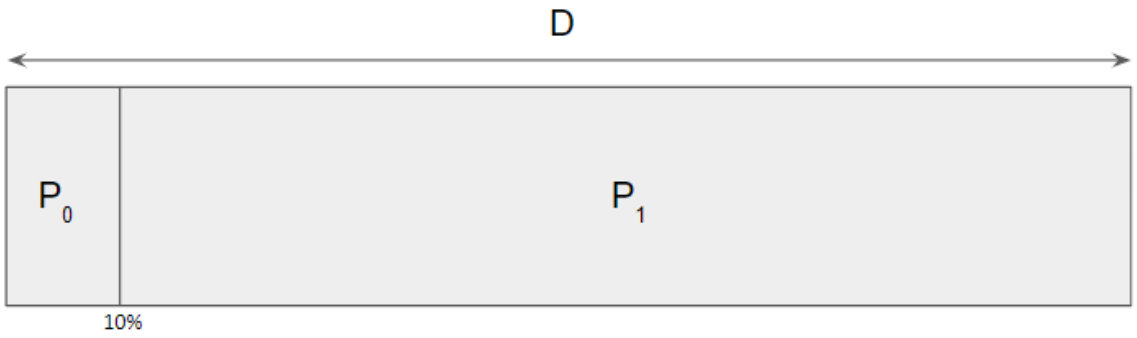


Figure 3.1: Bi-phase Data partition

$$D = \cup_{i=0}^{1} P_i$$

$$K_0 = Kmeans(P_0, centroids(random))$$
$$K_1 = Kmeans(D, centroids(K_0))$$

The idea is that with Qbeast we will be able to obtain small percentages of data faster than querying all the data, and "on the go" we will be able to perform a K-means execution with that smaller amount of data while we wait for the rest of the data to be loaded. Moreover, K-means standard initialization can be inefficient because random values are not optimal for center initialization, we can take advantage of the last state of the centers from a previous execution with an smaller percentage of the data, but still relevant. In any case, we can find many optimizations for the initialization of the centers. Our optimizations will always have the first phase where an initialization mode is needed, we could choose between random initialization or any kind of available optimization to set initial centers.0

To make it clearer, Figure 3.2 shows an example of how a data storage that supports efficient sampling behaves in time. The time spent to query percentages of data is strongly dependent on K, that represents the cost to query data. The total cost of the algorithm will depend on that value. For the simulation tests where we read data from disk, we consider that value as 0. Because we want to analyze the theoretical results with an ideal scenario we measure the run-times having all the data already cached. Though, with Qbeast its performance to query data (value k) will affect the overall cost of the algorithm. We can describe the cost of the algorithm f(Bi-phase) compared to the cost of the Normal K-means algorithm f(Normal) as follows:

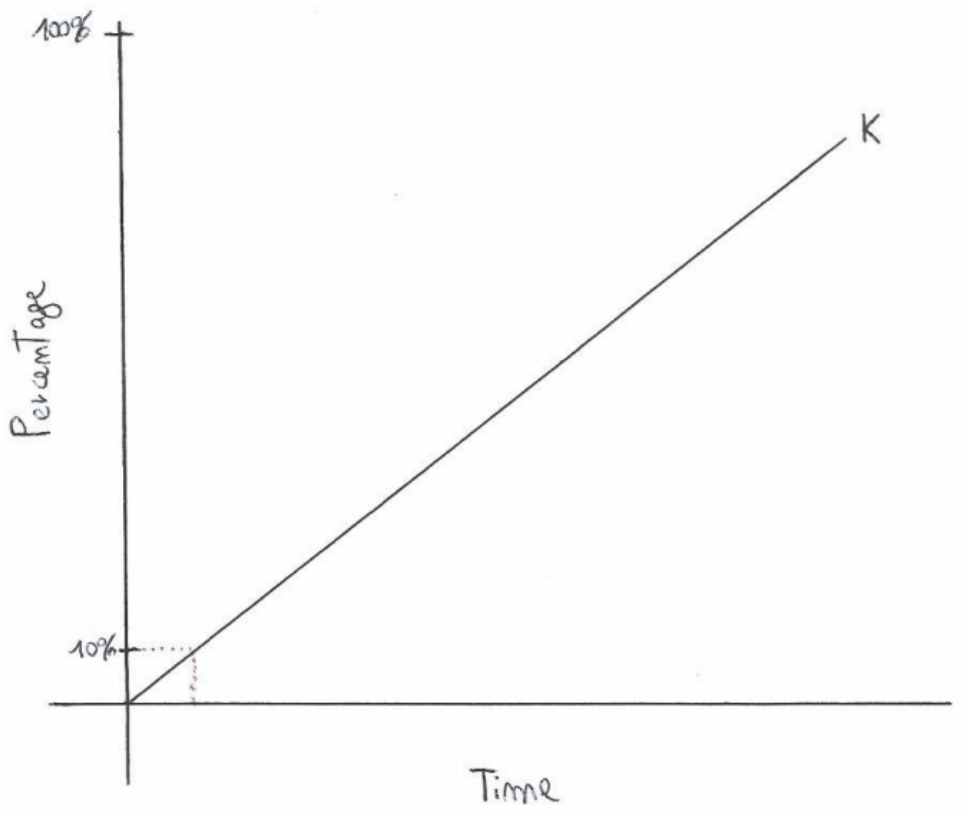


Figure 3.2: Bi-phase data percentage

$$f(Normal) = \frac{100}{k} + T_{km} \qquad f(Bi-phase) = \frac{10}{k} + max(\frac{90}{k}, T_{1phase}) + T_{2phase}$$

The code begins with arguments such as number of K-means centers (k), max iterations number and percentage of data to be sampled. Then 3.3, we create a resilient distributed data set [40] of the entire data with Spark connector to our Cassandra database, partition the data (i.e repartition = number of cores) and with Qbeast we are able to query the sample without processing all the data.

```
var sampledSource =
spark.read.format("org.apache.spark.sql.cassandra")
  .options(Map("keyspace" -> "benchmark", "table" -> "m4", "sampling" -> "true", "pushdown"
      -> "true"))
  .load().select("x", "y").rdd
  .map(row => Vectors.dense(Array(row.getDouble(0), row.getDouble(1))))
if (args.length > 6) sampledSource = sampledSource.repartition(args(6).toInt)
val sampledVector = sampledSource.sample(false, percentageData) //Sampling data
```

Figure 3.3: Bi-phase implementation: data sampling

Next, we initialize a K-means class with a previously saved initial model and execute the first phase. Finally, we execute the second phase with the output centers from the first one 3.4.

```
val oldRandomModel: KMeansModel = KMeansModel.load(spark.sparkContext, "randomModel")
val kmeansSample: KMeans = new KMeans().setK(numclust).setMaxIterations(iter)
.setSeed(3132626920565747559L).setInitialModel(oldRandomModel)
//Execute k0=Kmeans(P0,centroids(random)
val (timeSampling, modelcluster) = time(kmeansSample.run(sampledVector))
val allData = sampledSource
val meansRUN = new
    KMeans().setK(numclust).setMaxIterations(iter).setInitializationMode(mode)
.setInitializationSteps(steps).setInitialModel(modelcluster)
//Execute k1=Kmeans(D,centroids(k0)
val (timeFullData, clusterModelRandom) = time(meansRUN.run(allData))
```

Figure 3.4: Bi-phase implementation: K-means phases execution

3.1.2 Multi-phases

The point of this optimization is alike Bi-phase, to take advantage of Qbeast fast access to data over time, which requires smaller time to obtain small percentages of the data, and as percentage of data increases, more time is needed to obtain higher percentages of data. In this case, that can be visualized in Figure 3.5 along with the formal definition right below. N executions of K-means are performed, each execution uses a percentage of the data that increases over n, and each execution -except the first one- initializes its centers from the previous output. Precisely, we will test this optimization with three kinds of data percentage increase over the phases following the ideas from [25][12]:

- **Linear**: The first one starting at 10% of the data and increasing the same data percentage value(+10%) until reaching all data.
- **Geometric**: The second one starting at 10% of the data and increasing in powers of two until reaching all data (10%,20%,40%,80%,100%).
- **Order of magnitude**: Lastly we will test with smaller percentages in products of 10 (0,1%,1%,10%,100%). Which fits best when using Qbeast indexing to query data, since small percentages can be queried much faster.

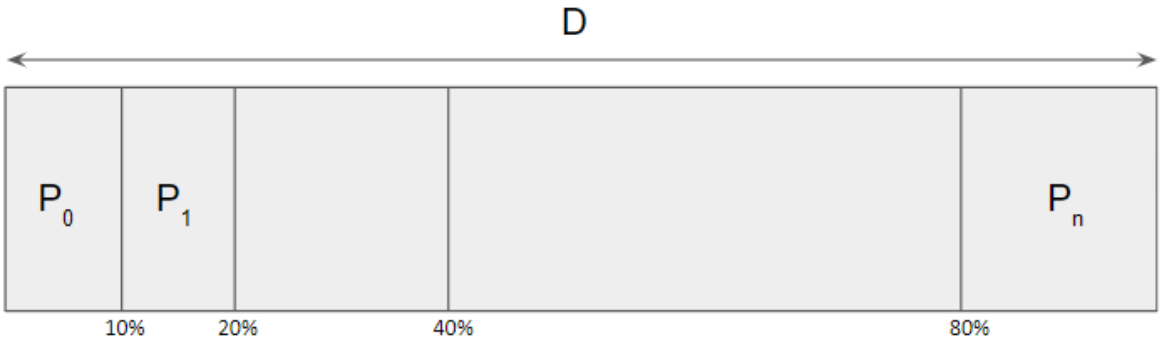


Figure 3.5: Multi-phases data partition

$$D = \cup_{i=0}^{n} P_i$$
$$\forall i, j | i \neq j \nexists x \in Pi \wedge x \in Pj$$

$$K_0 = Kmeans(P_0, centroids(random))$$
$$K_i = Kmeans(\sum_{j=0}^{i} P_j, centroids(K_{i-1}))$$
$$K_n = Kmeasn(D, centroids(K_{n-1}))$$

Yet again, the biggest benefit about implementing this optimization with efficient sampling is the ability to take advantage of smaller samples of the data while we wait for the rest of the data to be queried. Furthermore, in Figure 3.6 we have another graph showing the time spent to query different percentages of data, in the same way explained for Bi-phase. The cost of Multi-phases follows the same approach than Bi-phase but with more phases, and can be presented as f(Multi-phases):

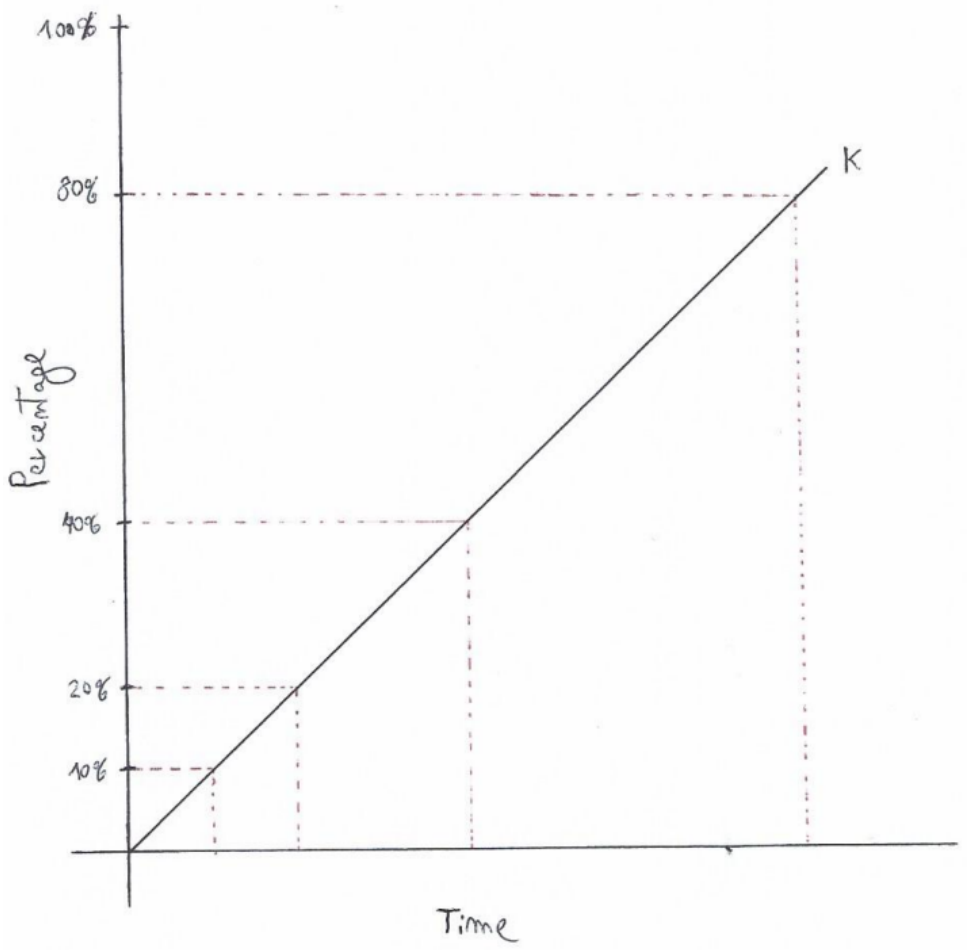


Figure 3.6: Multi-phases data percentages

$$f(Multi-phases) = \frac{10}{k} + max(\frac{10}{k}, T_{1phase}) + max(\frac{20}{k}, T_{2phase}) + max(\frac{40}{k}, T_{3phase}) + max(\frac{20}{k}, T_{4phase}) + T_{5phase}$$

Multi-phases optimization begins in the same way than Bi-phase, loading the data as a resilient distributed data set. Then, we initialize a K-means class as well as a K-means model (setting its value to NULL) so that we can iterate in a loop where we perform a K-means execution until reaching convergence for each phase, using the centers of the previous execution as initial solution. 3.7

```
var first = true
var timings = Array[Double]()
var modelCluster:KMeansModel = _
for (percentage <- percentageData) {
  if (first) {
    val sampledVector = sampledSource.sample(false, percentage)
    val (timing, a) = time(kmeans.run(sampledVector))
    modelCluster = a
    first = false
    timings = timings :+ timing
  }
  else {
    val sampledVector = sampledSource.sample(false, percentage)
    kmeans.setInitialModel(modelCluster)
    val (timing, b) = time(kmeans.run(sampledVector))
    modelCluster = b
    timings = timings :+ timing
  }
}
```

Figure 3.7: Multi-phases implementation: K-means phases execution

Chapter 4

Testing

To correctly test our assumptions, first we will validate that our optimizations provide good solutions. Then, we need a stable environment in which to run the tests. We will setup a Spark cluster at Marenostrum4 where we will also need to make sure that the tests are being executed correctly. Spark jobs can sometimes waste a lot of time when the configuration for a job isn't optimal because they do not take advantage of all available resources. For that, we will have to ensure that executions behave as expected. Moreover, because there will be a lot of time spent into configuring Spark cluster, we decided to perform two different groups of tests. The idea for each group of tests is the same, but the data source will be different. The first tests will be performed without efficient sampling, we will simulate by only timing the executions how we can query data with efficient sampling. Then, once we have analyzed these tests and understand the results we will analyze how we can take advantage of Multidimensional Indexing with efficient Sampling (MIS).

4.1 VALIDATION OF THE ALGORITHMS

We used silhouette score as the metric of choice for evaluating the clustering. Roughly, the silhouette score evaluates how well each point fits the cluster it was assigned to versus the next closest cluster. The values of the silhouette range from -1 to 1, where 1 represents perfect clustering.

With this, before performing the performance tests, we want to validate that the modifications that we have made in our optimizations do not affect negatively the results. We consider the output solutions to be as good as normal K-means if silhouette score of the outputs are very similarly against a normal K-means execution. We are not looking to get the perfect solution (score close to 1) for this validation test, we are only comparing scores.

In table 4.1, we can see that both of our optimizations show very similar silhouette score. Hence, our optimizations do not affect negatively the clustering solution.

	K-means	Bi-phase	Multi-phases
Silhouette score	0,506942	0,505206	0,505981

Table 4.1: Silhouette scores of the algorithms

A piece of the code to perform the validation test is shown in Figure 4.1. We used Python because the library sklearn includes the silhouette score algorithm. We implemented our optimizations in Python and performed the silhouette score for each algorithm.

```
def multiphases(dataset, n_clusters, multi_percentages):
    sample = dataset.sample(frac=multi_percentages[0])
    clusterer = KMeans(n_clusters=n_clusters, random_state=10)
    cluster_labels = clusterer.fit_predict(sample.values)

    for percentage in multi_percentages[1:]:
        sample = dataset.sample(frac=percentage)
        clusterer = KMeans(n_clusters=n_clusters, init=clusterer.cluster_centers_)
        cluster_labels = clusterer.fit_predict(sample.values)
    return cluster_labels

if __name__ == "__main__":
    n_clusters = 8
    bi_percentage = 0.01
    multi_percentages = [0.001, 0.01, 0.1]
    myData = pd.read_csv('datafile.csv')

    normal_labels = normalkmeans(myData, n_clusters)
    silhouette_normal = silhouette_score(myData.values, normal_labels)
    print("For normal Kmeans the average silhouette_score is :", silhouette_normal)
    bi_labels = biphase(myData, n_clusters, bi_percentage)
    silhouette_bi = silhouette_score(myData.values, bi_labels)
    print("For Bi-phase Kmeans the average silhouette_score is :", silhouette_bi)
    multi_labels = multiphases(myData, n_clusters, multi_percentages)
    silhouette_multi = silhouette_score(myData.values, multi_labels)
    print("For Multi-phases Kmeans the average silhouette_score is :", multi_labels)
```

Figure 4.1: Silhouette score test code

4.2 ENVIRONMENT SPARK ANALYSIS

4.2.1 Scalability in Spark

Spark application time is composed of two distinct parts, work done in the driver (alone) and work done in the executors (parallel). The ideal scenario is one where we have minimal work done in the driver, which consists mainly in synchronizing and communicating with the executors, because it stops all the executors work. The executors perform tasks that can be computed in parallel, the more time executors are performing tasks the better. In Figure 4.2, we can see an example of Spark's application time line. One of the simplest way to improve performance of spark application is to minimise the "do nothing" parts of the application. For example, when the application has not sufficient tasks and executors are idly waiting. We can do that by making sure that there will be enough tasks for the executors so that they have tasks ready to be computed as much as possible, also it is very important that we are not doing anything in the driver that can be parallelised and done in the executors. The work done in the driver should only consist in: file listing, split the computation, loading of hive tables and collect/take operations. Moreover, to have an scalable application in Spark there should be enough tasks, so that cores are used as much as possible. In the same way, once you have allocated more cores than tasks in any stage of a spark application, there won't be any further improvements in completion time.

There are many ways to control number of tasks, such as taking into account HDFS block size or setting the `spark.default.parallelism` parameter. In addition, task skew is not good for Spark, as seen in 4.2 unequal task skew means executors have trouble fitting many tasks together because their sizes differ.

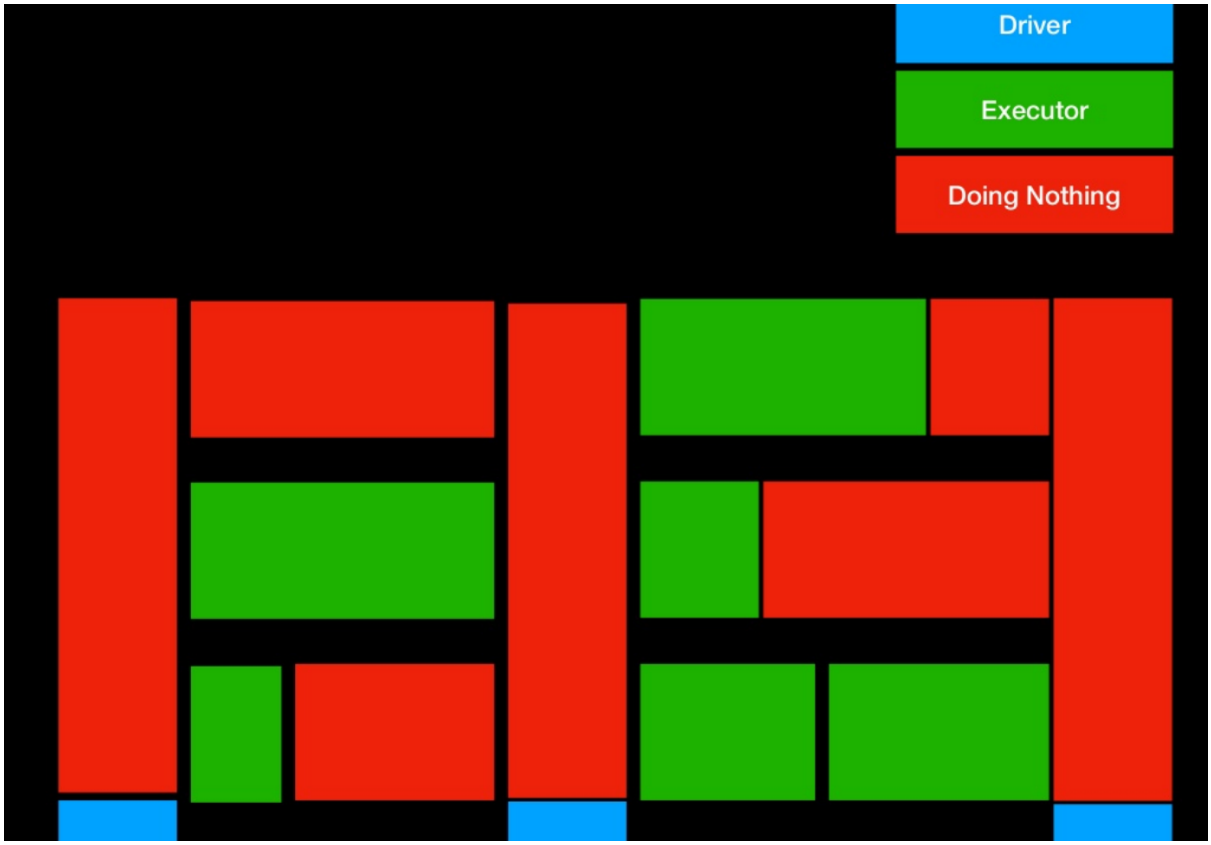


Figure 4.2: Spark application time line

Source: [35]

4.2.2 Scalability of the algorithms

We have performed several tests to analyze the scalability of the algorithms. For software, scalability can be referred to as parallelization efficiency, the ratio between the actual speedup and the ideal speedup obtained when using a certain number of processors. Ideally, we would like software to have a linear speedup that is equal to the number of processors (speedup = N), as that would mean that every processor would be contributing 100% of its computational power. Unfortunately, this is a very challenging goal for real application to attain.

On the one hand, In 1967, Amdahl pointed out that the speedup is limited by the fraction of the serial part of the software that is not amenable to parallelization [6]. Amdahl's law states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code. This is called strong scaling and we will perform some tests for the algorithms on this regard. Amdahl's law can be formulated as follows:
Speedup $= \frac{1}{(s + p/N)}$. Where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and N is the number of processors.

On the other hand, when a problem only requires a small amount of resources, it is not beneficial to use a large amount of resources to carry out the computation. A more reasonable choice is to use small amounts of resources for small problems and larger quantities of resources for big problems. Gustafon's law [21] was

proposed in 1988, and is based on the approximations that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the problem. It provides the formula for scaled speedup: $ss = s + p * N$. Where s, p and N have the same meaning as in Amdahl's law. With Gustafon's law the scaled speedup increases linearly with respect to the number of processors, and there is no upper limit for the scaled speedup. This is called weak scaling, where the scaled speedup is calculated based on the amount of work done for a scaled problem size (in contrast to Amdahl's law which focuses on fixed problem size).

4.2.3 Spark K-means implementation

In this section we explain the scalable K-means implemented in MLlib. The algorithm follows the same operations than K-means++ but specific modifications have been made to suit scalability. We will use the random initialization mode in the tests so we can jump directly to the second part of the algorithm.

Lloyd's iterations [28] carry out most of the computation of the algorithm, each iteration updates and propagates the new cluster centers and costs, the algorithm iterates until it either reaches max number of iterations or the algorithm converges because further iterations would not improve the solution of the output centers. A pseudo-code can bee seen in Figure 4.3. For each iteration we invoke the *collect* function of Spark, which forces the algorithm to communicate the workers with the driver, so that the all the centers are correctly synchronized between the workers. For instance, imagine that the workers are divided spatially in partitions in a 2D space, if we don't communicate the cluster centers that do are not included in a worker partition we could not run the algorithm in a distributed way.

```
// Execute iterations of Lloyd's algorithm until converged
while (iteration < maxIterations && !converged) {
    // Find the new centers
    val collected = data.mapPartitions { points =>
    // Collect function that requires synchronization
    }.collectAsMap()
    // Update the cluster centers and costs
    converged = true
    newCenters.foreach { case (j, newCenter) =>
    if (converged &&
      !distanceMeasureInstance.isCenterConverged(centers(j), newCenter, epsilon)) {
      converged = false
    }
    centers(j) = newCenter
    iterations += 1
}
```

Figure 4.3: Spark K-means Lloyd's iterations

To make it clearer, Figure 4.4 shows the communication process that needs to be carried out in each iteration

of the algorithm. We can deduce that this communication affects strongly the performance of the algorithm and may induce in an overhead when having a high number of centers and/or data points.

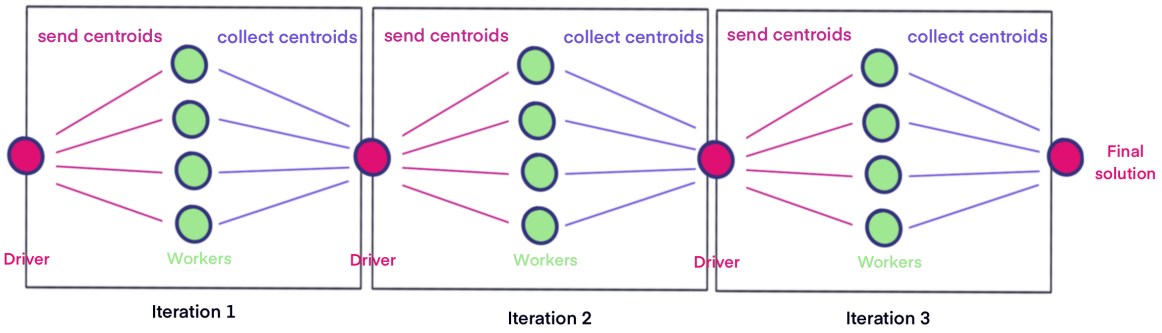


Figure 4.4: Spark K-means driver-executors communication

4.2.4 Apache Spark tuning

Spark supports three types of clusters:

- Standalone: meaning Spark will manage its own cluster.
- YARN: using Hadoop's YARN resource manager.
- Mesos: Apache's dedicated resource manager project.

According to [26], it is recommended to start with a standalone cluster if this is a new deployment. Since Standalone mode is the easiest to set up and will provide almost all the same features as the other cluster managers, we decided to configure Spark in standalone mode [4]. It is very important that every application correctly configures the number of cores for executors and how much memory allocate for each one. In a Spark standalone deployment the hierarchy is like the following one:

```
|------------|
|   Worker   |  1 per server
|------------|
      |
     \ /
|------------|
|   Executor |  1 process/JVM. You can configure how many CPU ( **spark.executor.cores** ) and
|------------|  RAM (**spark.executor.memory**) assign to each Executor
      |
     \ /
|------------|
|   Task     |  The single Thread/Operation executed in a spark graph. Usually, you have 1 CPU for TASK
|------------|  but it can be changed with **spark.task.cpu**
```

Figure 4.5: Spark hierarchy in standalone mode

4.2.4.1 Tuning Resource Allocation

Because we have 48 cores and 98GB of memory per node, the first impulse would be to use **47 cores** and **98GB of memory** having 1 executor per node. However, this is the wrong approach because:

- We must leave at least 1 core per socket for the Linux Kernel.
- We must leave some RAM for the Linux Kernel, 98GB + the executor memory overhead won't fit within the 98GB capacity of the node
- We must leave some RAM for the OS paging.

A better option would be to configure spark to use **11 cores per executor** and **22GB of memory** per executor. Because each node in MareNostrum4 has 98GB of memory and 48 cores, in this way we create 4 executors per node, and we leave 4 CPUs 10G of memory free for the OS. Although this approach is expected to give good results, we have decided to reserve an entire node for the driver to be 100% sure that the master node is never stressed and the executors aren't penalized for that. We followed these guides [5][3] to set up the optimal number of executors per core, that goes between 3 and 5 according to the guides. This is an example of the script that we used to set up a Spark cluster 4.6.

```
#!/bin/bash
module load java/8u201
HOSTS=(‘scontrol show hostnames $SLURM_HOSTS‘)
export SPARK_SLAVES=/tmp/slaves.$SLURM_JOB_ID
scontrol show hostnames $SLURM_HOSTS|grep -v ${HOSTS[0]}|awk ’{print $0"-ib0"}’ >$SPARK_SLAVES
echo using slaves
export SPARK_WORKER_CORES=42
export SPARK_MASTER_HOST=${HOSTS[0]}-ib0
SPARK_HOME=/gpfs/projects/bsc31/SPARK_MN4/2.4.1/spark/
srun cp -r /gpfs/projects/bsc31/SPARK_MN4/2.4.1/spark/ /scratch/tmp/
SPARK_HOME=/scratch/tmp/spark
PATH=$SPARK_HOME/bin/:$PATH
ssh  $SPARK_MASTER_HOST "$SPARK_HOME/sbin/start-master.sh -h $SPARK_MASTER_HOST"
echo started master
sleep 30
for h in ‘cat $SPARK_SLAVES‘
do
  echo starting slave in $h
  ssh $h "$SPARK_HOME/sbin/start-slave.sh -h $h spark://$SPARK_MASTER_HOST:7077"
done
echo started slaves
sleep 10000000
or
#Here we could add spark-submit
```

Figure 4.6: Example of a script to set up Spark cluster at Marenostrum4

4.2.4.2 Tuning Parallelism

In order to understand the scalability of K-means and configure the Spark environment correctly we have used artificial data sets, the data sets are shown in table 4.2, for these data sets, each data point has 4 dimensions.

	Data points	CSV size
1M.csv	1.000.000	74M
10M.csv	10.000.000	736M
100M.csv	100.000.000	7.2G
1000M.csv	1.000.000.000	72G

Table 4.2: Synthetic data sets

To tune Spark's parallelism we performed some tests and analysed through Spark's user interface how the executors were behaving, in terms of stress workload to try to distribute tasks in a more efficient way and detect whether the nodes were being correctly balanced or not. We realised that the nodes workloads were too disparate probably because we were using heavy data sets and a lot of I/O calls were performed. One of

the parameters that Spark uses to solve I/O bottlenecks and in other words, improve the parallelism for the application is the value of `spark.shuffle.file.buffer`[2] which is set to 32KB by default, but the block size in MareNostrum4 (GPFS Distributed Parallel File System) is 16MB, we changed this parameter to be equal to MareNostrum4's GPFS File Block size and we improved the parallelism of the execution and hence the run-time.

By Default, the File Block size influences the number of tasks leading to the creation of a task per block. However, in a larger setting we need more tasks to use all the cores. Therefore, we decided to set via software the task partition value from code, to be equal to the number of cores used. This configuration showed better performance and made more sense when increasing the number of cores for the scalability tests.

4.2.5 K-means performance metrics

In this section we describe the methodology followed in order to make sure that the tests are being performed correctly and so that we are able to understand the results and scalability of the algorithm.

One of the main tools that allowed us to analyze the performance of our tests was Sparklens[1], which is a profiling tool for Spark with built-in Spark Scheduler simulator. It helped us to understand how many computing resources are lost in the driver, the scalability limits of spark applications, as well as how efficiently is a given spark application using the compute resources provided to it.

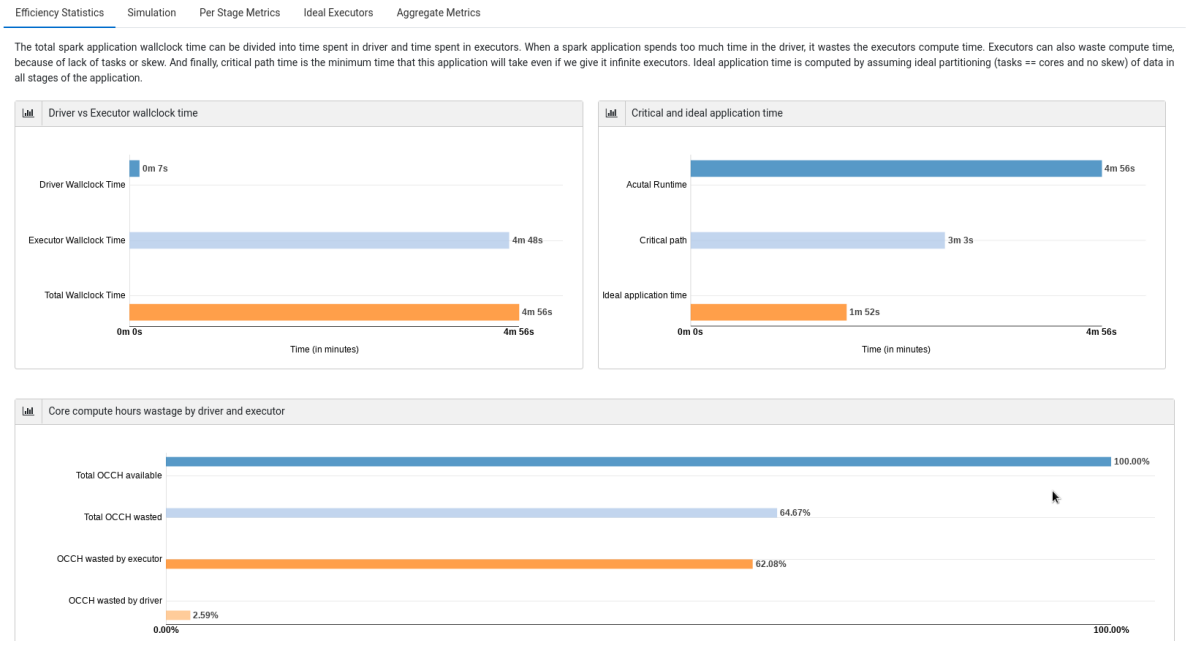


Figure 4.7: Sparklens HTML page

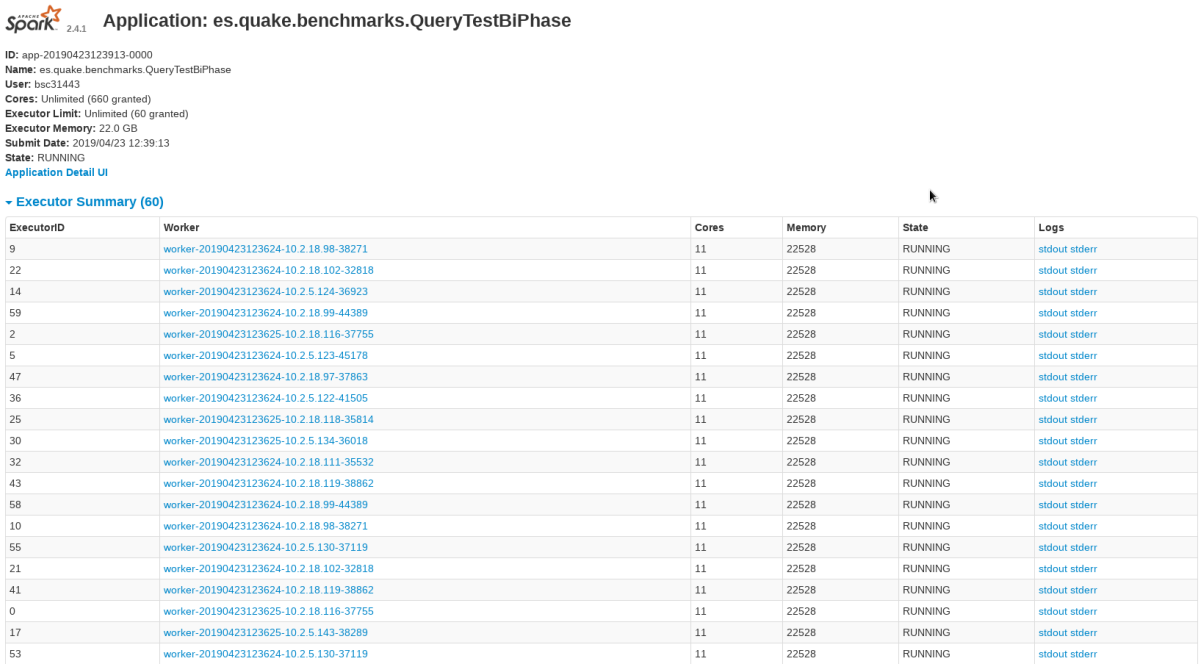
As it can be seen in figure 4.7, there are different efficiency metrics shown in Sparklens:

- **Driver vs Executor wallclock time** which shows the time spent by the driver and executors, along with the total time of the application, which is the sum of both.

- **Critical and ideal application time** compares the actual runtime against the Critical path and ideal application time. The critical path follows that minimum time take by spark application irrespective of the number of executors, can be calculated as: *Time spent in the driver + SUM (for each stage, time spent in the slowest task)*. The main idea is that we cannot make the driver run faster with more executors and we cannot make the slowest task of any stage run faster with more executors. In other words, once you have allocated more cores than tasks in any stage of a spark application, you will not see any further improvements in completion time because any additional core doesn't have any task to run. Moreover, an ideal Spark application is one which has minimal skew, high parallelism and does minimal work in driver. An ideal Spark application is composed of ideal stages which their wall clock time can be computed as: *Sum of time spent by each task in a stage divided by cores available to the stage.* An ideal stage uses all the cores at all the times.
- **Core compute hours wastage by driver and executor** Describes the CPU efficiency by showing the percentage of time wasted by the driver and executor in One Core Compute Hours (OCCH): Measure of total compute power available from cluster. One core in the executor, running for one hour, counts as one OneCoreComputeHour, Executors with 4 cores, will have 4 times the OneCoreComputeHours compared to one with just one core. Similarly, one core executor running for 4 hours will OnCoreComputeHours equal to 4 core executor running for 1 hour.

Spark also has an interface that can be accessed on port 8080 when running an application, there it presents details in a user interface with Key Performance Indicator (KPI) parameters. In our case, to access the Spark's UI we had to user the url -> *NodeName:8080*. To bypass the Marenostrum Security Policy, we use the **ssh** built-in proxy capability.

In Figure 4.8 the main page of Spark's UI is shown where a list of the executors and their tasks is shown. The most useful page for us to analyze the performance of K-means was the executors information section in application detail UI.



Application: es.quake.benchmarks.QueryTestBiPhase

ID: app-20190423123913-0000
Name: es.quake.benchmarks.QueryTestBiPhase
User: bsc31443
Cores: Unlimited (660 granted)
Executor Limit: Unlimited (60 granted)
Executor Memory: 22.0 GB
Submit Date: 2019/04/23 12:39:13
State: RUNNING
Application Detail UI

Executor Summary (60)

ExecutorID	Worker	Cores	Memory	State	Logs
9	worker-20190423123624-10.2.18.98-38271	11	22528	RUNNING	stdout stderr
22	worker-20190423123624-10.2.18.102-32818	11	22528	RUNNING	stdout stderr
14	worker-20190423123624-10.2.5.124-36923	11	22528	RUNNING	stdout stderr
59	worker-20190423123624-10.2.18.99-44389	11	22528	RUNNING	stdout stderr
2	worker-20190423123625-10.2.18.116-37755	11	22528	RUNNING	stdout stderr
5	worker-20190423123624-10.2.5.123-45178	11	22528	RUNNING	stdout stderr
47	worker-20190423123624-10.2.18.97-37863	11	22528	RUNNING	stdout stderr
36	worker-20190423123624-10.2.5.122-41505	11	22528	RUNNING	stdout stderr
25	worker-20190423123625-10.2.18.118-35814	11	22528	RUNNING	stdout stderr
30	worker-20190423123625-10.2.5.134-36018	11	22528	RUNNING	stdout stderr
32	worker-20190423123624-10.2.18.111-35532	11	22528	RUNNING	stdout stderr
43	worker-20190423123624-10.2.18.119-38862	11	22528	RUNNING	stdout stderr
58	worker-20190423123624-10.2.18.99-44389	11	22528	RUNNING	stdout stderr
10	worker-20190423123624-10.2.18.98-38271	11	22528	RUNNING	stdout stderr
55	worker-20190423123624-10.2.5.130-37119	11	22528	RUNNING	stdout stderr
21	worker-20190423123624-10.2.18.102-32818	11	22528	RUNNING	stdout stderr
41	worker-20190423123624-10.2.18.119-38862	11	22528	RUNNING	stdout stderr
0	worker-20190423123625-10.2.18.116-37755	11	22528	RUNNING	stdout stderr
17	worker-20190423123625-10.2.5.143-38289	11	22528	RUNNING	stdout stderr
53	worker-20190423123624-10.2.5.130-37119	11	22528	RUNNING	stdout stderr

Figure 4.8: Spark's UI

In Figure 4.9 we can see several parameters about the current workloads of the executors for the application that is being run. Starting off with the most obvious ones, it is very important that the application has no failed tasks, otherwise the executors are wasting quite a lot of time repeating tasks, if there was the case where a task failed, we could check the logs of the executor that failed and check what went wrong. Another thing that we want to analyze in 4.9 is whether the executors are sharing the total workload in a balanced way. We can analyze this by looking at the Task time for the executor with most task time against the one with less task time, and check that the difference isn't high. Along with the same line of thought, we can check that the difference of the higher value of Total Tasks, Input and Storage Memory for the executors against the lower of each isn't significant. Low values on these operations mean that our application is being distributed through the nodes efficiently.

As a further matter, we have run into some challenges regarding Java Virtual Machine(JVM). Depending on the data set that we were using the application was stressing JVM heap space inducing into large Garbage Collector(GC) times. Java Virtual Machine Garbage Collector takes care of the process of reclaiming the run-time unused memory automatically. In other words, it is a way to destroy the unused objects. We have used VisualVM, which is a tool that provides a visual interface to visualize detailed information about java applications that are being executed on a JVM, and to solve problems profiling these applications. Of course having large GC times is not optimal for benchmarking. VisualVM 4.10 allowed us to visualize the heap space usage while performing experiments, which was being stressed because of bad task partition that we solved when tuning the parallelism of Spark application.

Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(61)	1642	277.2 MB / 745 GB	0.0 B	660	621	0	84607	85228	1.9 h (3.2 min)	55.5 GB	0.0 B	418.4 KB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(61)	1642	277.2 MB / 745 GB	0.0 B	660	621	0	84607	85228	1.9 h (3.2 min)	55.5 GB	0.0 B	418.4 KB	0

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
0	10.2.18.116:43736	Active	22	3.9 MB / 12.4 GB	0.0 B	11	11	0	1133	1144	1.8 min (3 s)	754.6 MB	0.0 B	0.0 B	stdout stderr	Thread Dump
driver	s06r2b49:42496	Active	0	644.2 KB / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	10.2.18.116:37589	Active	44	7 MB / 12.4 GB	0.0 B	11	11	0	2266	2277	2.2 min (3 s)	1.5 GB	0.0 B	0.0 B	stdout stderr	Thread Dump
2	10.2.18.116:37524	Active	44	7.1 MB / 12.4 GB	0.0 B	11	11	0	2266	2277	2.1 min (4 s)	1.5 GB	0.0 B	0.0 B	stdout stderr	Thread Dump
3	10.2.18.116:42031	Active	22	3.8 MB / 12.4 GB	0.0 B	11	11	0	1133	1144	1.8 min (4 s)	743 MB	0.0 B	0.0 B	stdout stderr	Thread Dump
4	10.2.5.123:45113	Active	22	3.9 MB / 12.4 GB	0.0 B	11	11	0	1133	1144	1.8 min (3 s)	751.1 MB	0.0 B	0.0 B	stdout stderr	Thread Dump
5	10.2.5.123:38481	Active	22	3.9 MB / 12.4 GB	0.0 B	11	11	0	1133	1144	1.9 min (3 s)	747.5 MB	0.0 B	0.0 B	stdout stderr	Thread Dump
6	10.2.5.123:46024	Active	22	3.8 MB / 12.4 GB	0.0 B	11	11	0	1133	1144	1.8 min (3 s)	738.4 MB	0.0 B	0.0 B	stdout stderr	Thread Dump
7	10.2.5.123:39536	Active	22	3.9 MB / 12.4 GB	0.0 B	11	11	0	1133	1144	1.8 min (3 s)	745.5 MB	0.0 B	0.0 B	stdout stderr	Thread Dump
8	10.2.18.98:39557	Active	36	5.9 MB / 12.4 GB	0.0 B	11	11	0	1854	1865	2.0 min (3 s)	1.2 GB	0.0 B	0.0 B	stdout stderr	Thread Dump
9	10.2.18.98:44549	Active	36	5.8 MB / 12.4 GB	0.0 B	11	11	0	1854	1865	2.0 min (3 s)	1.2 GB	0.0 B	0.0 B	stdout stderr	Thread Dump

Figure 4.9: Executors information on Spark's UI

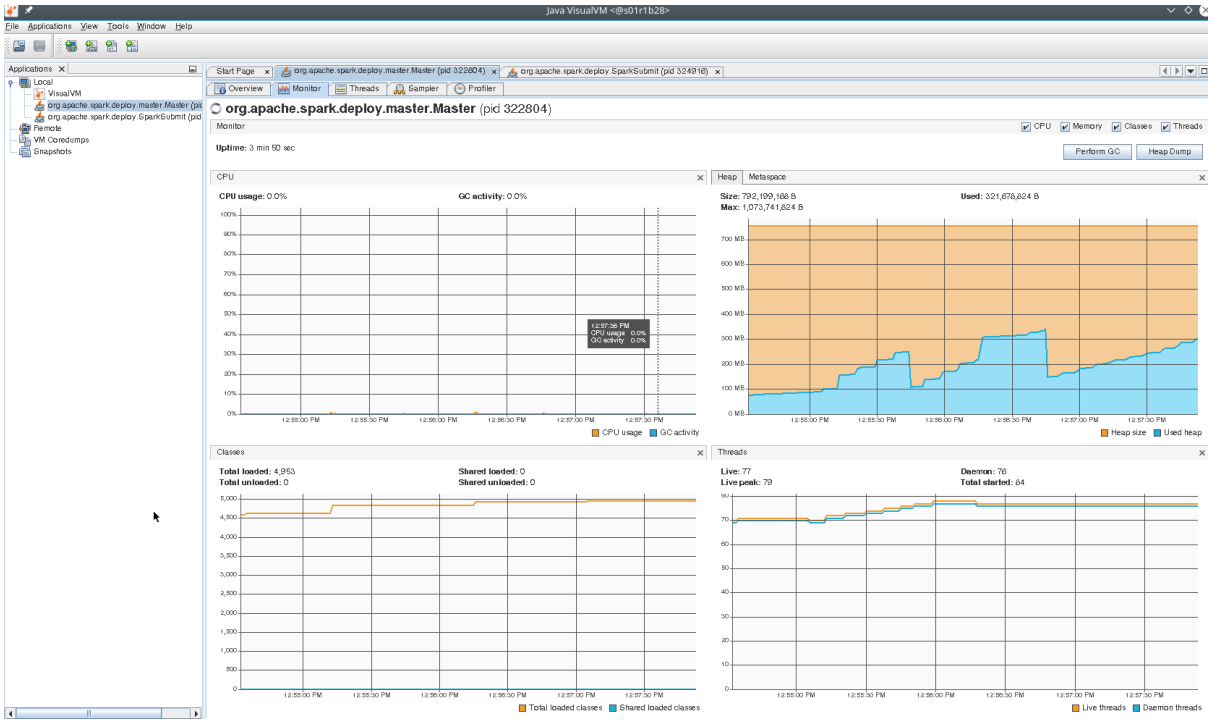


Figure 4.10: Java VisualVM interface

4.3 BENCHMARK TEST RESULTS

The results of the different tests are described in this section. First we test K-means scalability, then we test the optimizations and finally we analyze how we can improve with MIS (multidimensional indexing efficient sampling).

4.3.1 Input data

For the tests on K-means scalability and the optimizations we have used data sets contained with real data: geo-tagged market transactions obtained from Open Street Map [38].

In table 4.3, you can see the different files containing the data sets that we have used, with their data points and sizes, each data point having 2 dimensions.

	Data points	CSV size
40M_set.csv	40.000.000	2.6G
400M_set.csv	400.000.000	26G
3-2MM_set.csv	3.200.000.000	174G

Table 4.3: Geo-tagged data sets

For the tests related to weak scalability we have used the files shown in table 4.4.

	Data points	CSV size
weak1	100.000.000	6,4G
weak2	200.000.000	13G
weak4	400.000.000	26G
weak8	800.000.000	52G
weak16	1.600.000.000	105G

Table 4.4: Skewed geo-tagged data sets

4.3.2 Queries

We have tested K-means and we have compared run times against our optimizations with different data set sizes and configurations. The different data set sizes are shown in table 4.3, and we have tested using the following set of nodes in MareNostrum4: {16, 8, 4, 2, 1}. We have implemented three different queries to test:

- **Normal K-means:** As a normal execution of K-means for comparison.
- **Bi-phase:** As the first proposed algorithm.
- **Multi-phases:** As the second proposed algorithm.

4.3.3 Environment specification

We will be performing our tests in the general-purpose block of MareNostrum4 with a different set of nodes. Each node has 98GB of memory and two *Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz* chips, each with 24 processors for a total of 48 cores per node. For our tests we setup the Spark environment with a script that uploads all the necessary resources for the Spark cluster to be able to run our tests.

4.3.4 Metrics specification

We use the same random initial solution in all executions, and we maintained the same seed for initial centers across all tests because having different random initial centers hinders the process to obtain results. The metrics that we will take into account to evaluate the results are:

- Run-time measured in seconds.
- Number of iterations until K-means convergence.
- Speedup over nodes.
- Scalability loss over nodes.

4.3.5 K-means scalability tests

As we want to compare our optimizations against a normal execution of K-means, we have also performed several tests of K-means in the same environment.

4.3.5.1 Hard scalability

First, we want to test how it does not scale well. We have performed a set of strong scaling tests to observe how the scalability of the algorithm behaves as we increase the number of executors. We have performed the tests in two groups, one having the OS block size (block size = 16MG) as the parameter for task re-partition and the other one using sparkContext in the code to re-partition the tasks (repartition = number of cores). All these tests have been performed twice, as we have tested using two data sets 2,6G and 26G.

Because we wanted a more precise observation of the scalability of the algorithm, this time instead of scaling over nodes we have scaled over the executors. Remember that we have 4 executors per node and each node has 44 CPUs, thus each executor has 11 CPUs. Taking that into account we have performed the tests with 1, 2, 4, 8, 16, 32 and 64 executors. In graph 4.11, we can see that once we reach a number of executors equal to 8/16, there are no speedup improvements even if we increase the number of executors. We have a file block size = 16MB, the data set size is 2,6G. Then: $\frac{2600\text{MB}}{16\text{MB/Task}} = 162.5\text{Task}$, with 16 executors we have 174 cores which is higher than the task partitioning that we have configured, thus there are no speedup improvements when the number of executors is higher than 16.

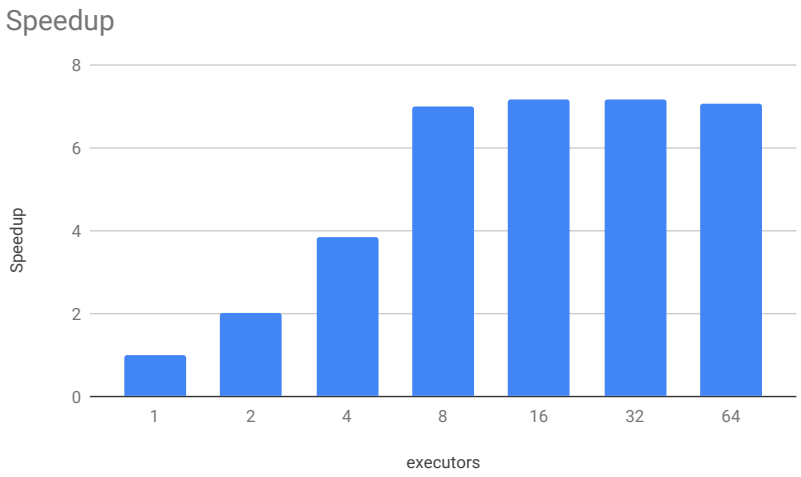


Figure 4.11: Speedup Normal K-means strong scaling, repartition = block size, data size=2,6G

Moreover, in graph 4.12 we are comparing the scalability loss over the executors, again with a data set size of 2,6G and task partitioning equal to file block size. The results show that as we increase the number of executors the scalability loss increases. With these results it is obvious that the algorithm doesn't scale well.

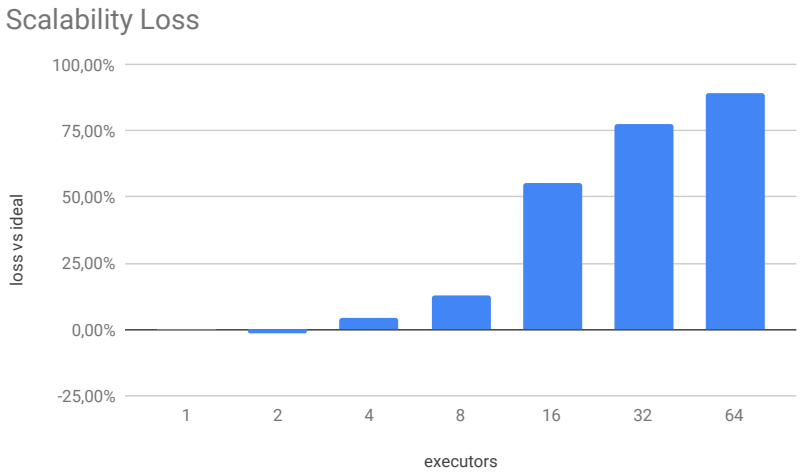


Figure 4.12: Scalability Loss Normal K-means strong scaling, repartition = block size, data size=2,6G

The same tests have been performed with a higher data set of 26G, in graph 4.13 we can see that because we have more data the amount of tasks is higher, and for that better speedups are achieved than before when we used a smaller data set. In this case we have enough tasks for all the executions so that we always have pending tasks to be allocated between the cores.

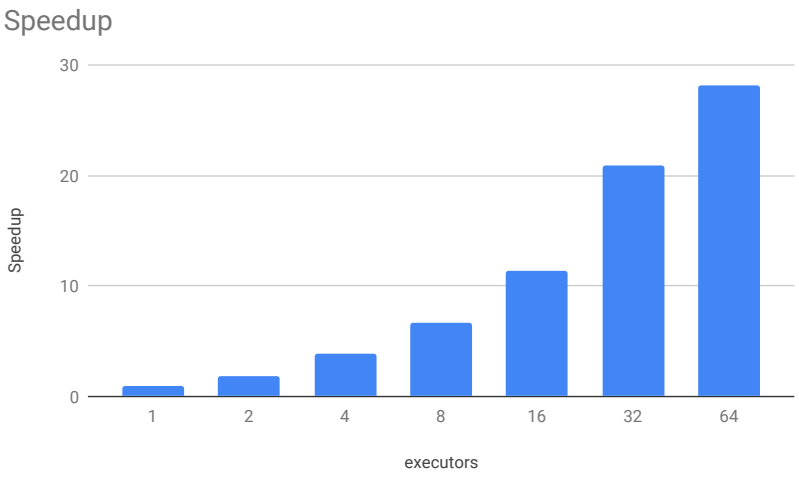


Figure 4.13: Speedup Normal K-means strong scaling, repartition = block size, data size=26G

Though, in graph 4.14 where data size is 26G, the scalability loss against the number of executors still shows bad scalability for the algorithm as there is a significant overhead when using more resources.

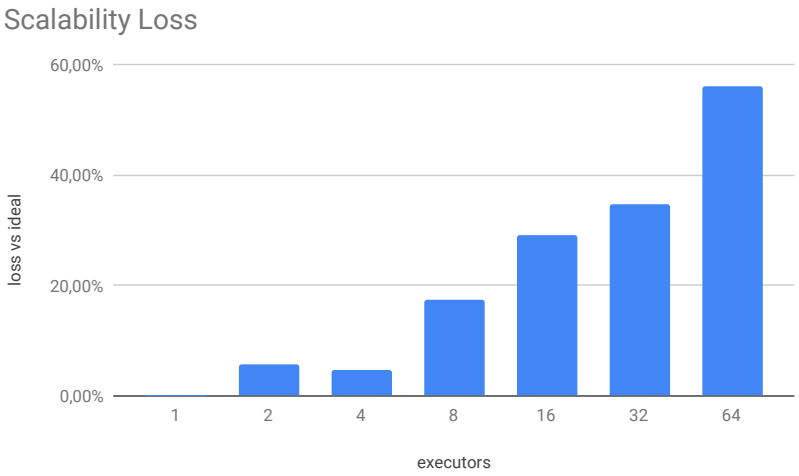


Figure 4.14: Scalability Loss Normal K-means strong scaling, repartition = block size, data size=26G

Now that we have observed that the algorithm doesn't scale well, we want to analyze how it behaves with a different task partitioning approach. Before, we were using the same number of task partition for each execution, now we will increase the number of tasks at the same rate that we increase the number of cores, we are also testing strong scalability of the algorithm. The graph 4.15 shows similar results. Although better run-times in general are achieved, once we reach 16 executors there are no scalability improvements.

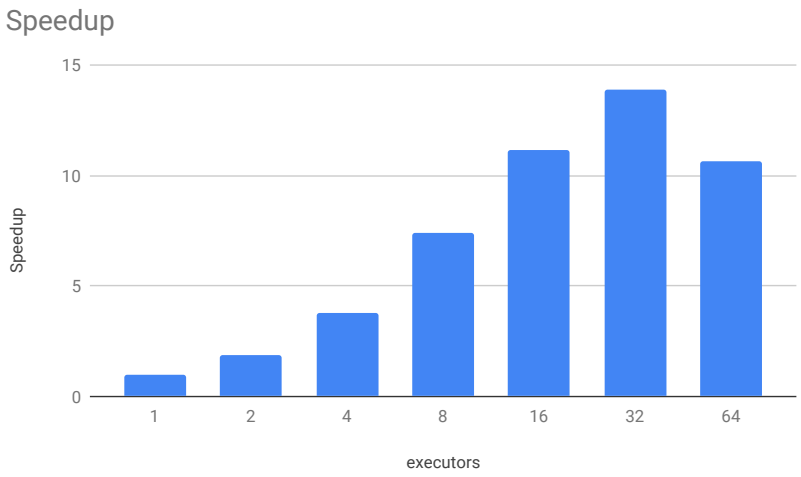


Figure 4.15: Speedup Normal K-means strong scaling, repartition = number of cores, data size=2,6G

This graph 4.16 presenting the Scalability loss of the algorithm, with data size of 26G and task partitioning configured via software, it shows that the algorithm gets worse in terms of scalability even though we increased the number of tasks in relation to the number of cores that we have at disposal. This is due by the fact that the smaller the tasks are, the higher overhead price we are paying, due to the additional synchronization and communication between the Driver and the Workers.

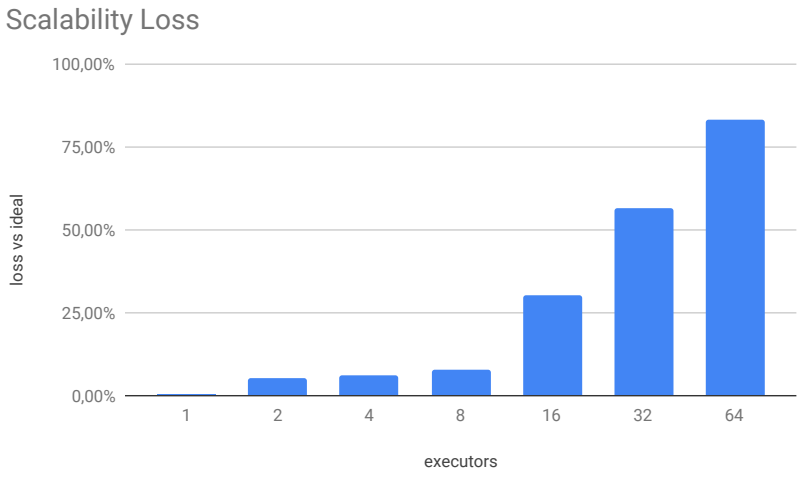


Figure 4.16: Scalability Loss Normal K-means strong scaling, repartition = number of cores, data size=2,6G

Defining the number of partitions from code has shown better results overall, hence we will use this approach for the rest of the tests. An overall comparison of the speedups is shown in graph 4.17 along with the ideal speedup (orange color).

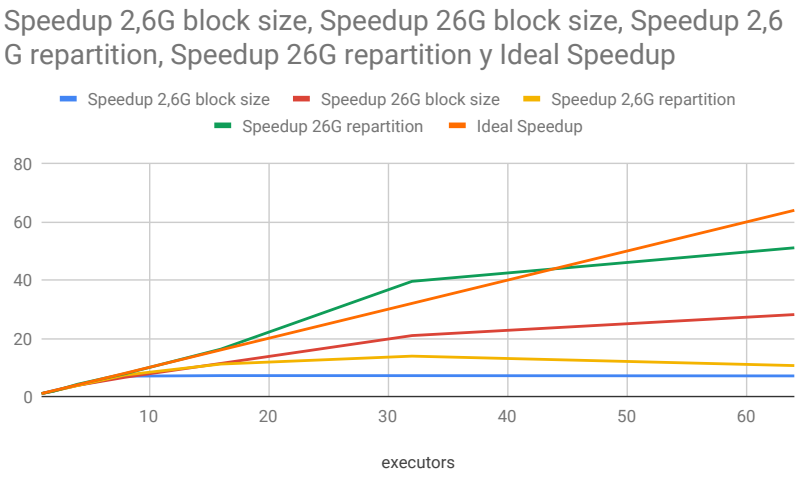


Figure 4.17: Speedup comparison

4.3.5.2 Weak scalability

Moving on, a test has been performed to analyze the weak scalability of the algorithm. We have measured the speedup, time and number of iterations based on the amount of work done for a scaled problem size. The input data is shown in 4.4, where each file is used with a set amount of resources, increasing the amount of resources jointly with the size of the files. The results at 4.18 show that the time and number of iterations are consistently similar as the amount of work scales with the problem size, except for the distinctive result in the last execution where the size is 105G. This is consistent with the previous tests, the last test with 64 executors (16 nodes) has overreached the threshold where the scalability of the algorithm is not making improvements. This is because once reaching a certain amount of data the driver becomes a bottleneck when communicating and synchronizing with the Worker nodes.

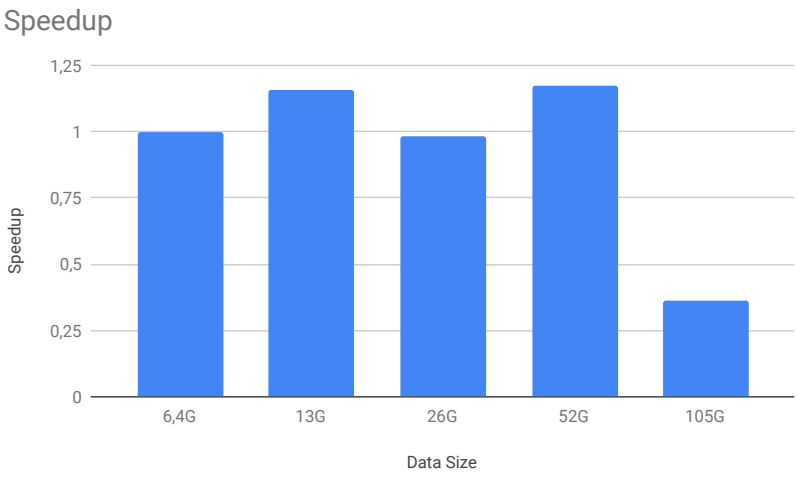


Figure 4.18: Speedup Normal K-means weak scaling, repartition = number of cores

In graph 4.19, we present the times and iterations over executors for the weak scaling test, we have separated the Y axis in two to represent it in a clearer way. In the same way, the last execution is distinct from the previous ones as it consumes much larger times and iterations remarking that large sizes of data induce into a driver bottleneck. The X axis values of the graphs, data sizes and number of nodes are complementary: {1, 2, 4, 8, 16} number of nodes -> {6,4G, 13G, 26G, 52G, 105G} data sizes.

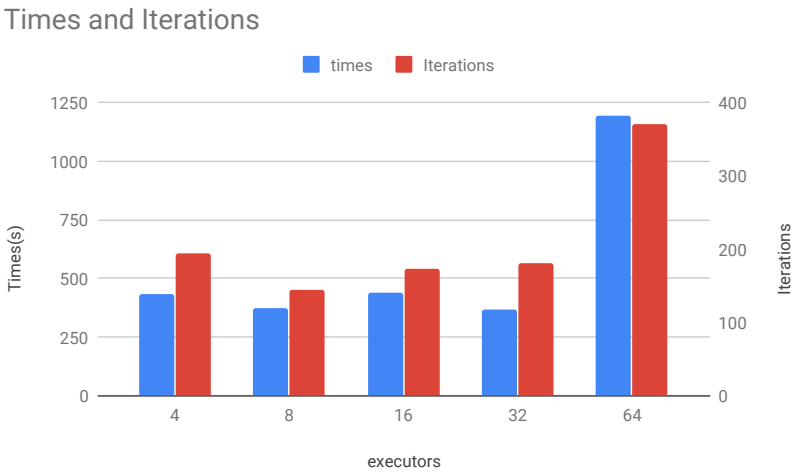


Figure 4.19: Scalability Loss Normal K-means weak scaling, repartition = number of cores

4.3.6 Optimization tests

For these tests, we decided to perform them measuring the time and iterations of only K-means executions, ignoring the time spent to read and sample data. The reasoning behind this is that we want to first validate the hypothesis of our optimizations, loading the data from disk is very different than loading from Cassandra on top of Qbeast, which is expected to take little time when reading small percentages of data. Though, we also present some results taking into account both read time and execution time in the Efficient data sampling section.

4.3.6.1 Bi-phase

For Bi-phase optimization, first we wanted to test different values for the amount of percentage data to be sampled. The first phase samples a portion of the data, so we need to define a value. we tested with different percentages: {0.1%,1%,10%,20%,30%,40%} of the data and got the results shown in graph 4.20. We are comparing run-times and iterations of different sampled percentages for Bi-phase executions, with a data set size of 26G and 8 nodes for all executions. As it can be seen 10% percentage got the best results forming the inflection point. Even though 0.1% and 1% are very low percentages they achieve good run-times, thus when taking into account data load times it could be beneficial to use lower percentages. Then, from 10% onward the algorithm shows even worse run-times. Hence, we chose 10% value as the percentage parameter for Bi-phase executions and comparison with the other algorithms.

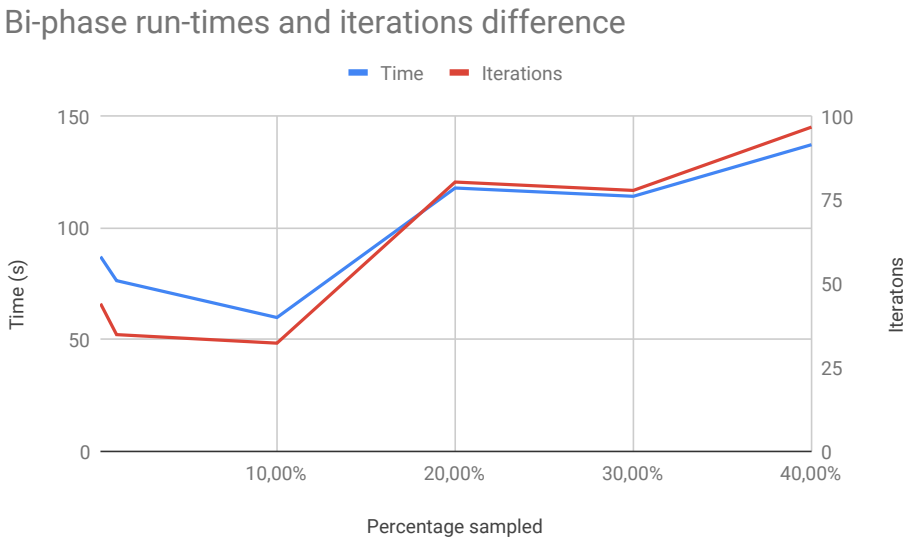


Figure 4.20: Bi-phase run-times and iterations with different percentages, size=26G

The trade-off of this approach is that with an initial small sample it is faster to get an initial solution, but as it is a coarser approximation we will need to perform iterations on the full data set to converge.

For both Bi-phase and Multi-phases tests we have measured the time and iterations in the following way:

- **Time:** The sum of times for all phases.
- **Iterations:** The sum of iterations multiplied by the percentage of data being computed. For instance, in Bi-phase if we get 115 iterations in the first phase, where 0.1 percentage of data has been sampled, in this phase we measure 115*0.1=11.5 iterations, if the last phase costs 51 iterations we will have a total of 62.5 iterations.

Going back to testing, we performed a strong scaling test with nodes {1,2,4,8,16}, having 44 cores and 4 executors per node. The results shown in graph 4.21 present better run-times than normal K-means executions, which will be discussed later in comparison with the other algorithms. Though, scalability issues persist once reaching 8/16 nodes.

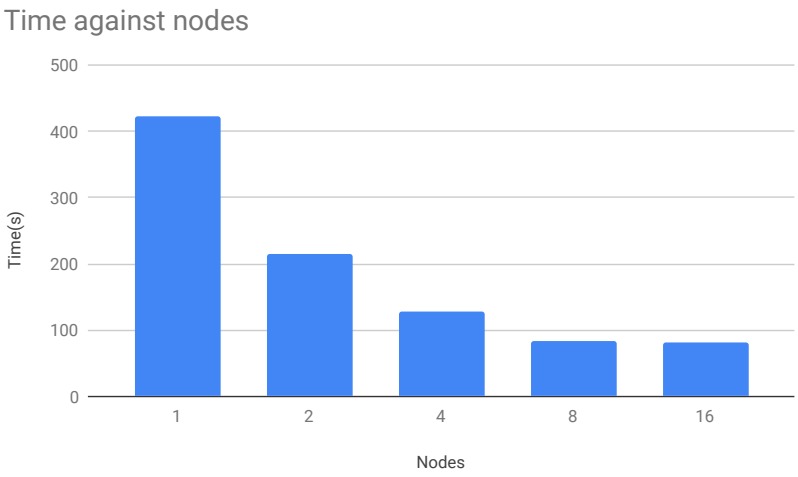


Figure 4.21: Bi-phase run-times over nodes, data size=26G

Furthermore, the same weak scalability test performed for Normal K-means has been performed for Bi-phase implementation. In figures 4.22 4.23 we can observe that the time and number of iterations are consistently similar as the amount of work scales with the problem size 4.4, before reaching the threshold where the scalability of the algorithm is not making improvements with 16 nodes. It's a very similar result than the Normal K-means weak scaling test.

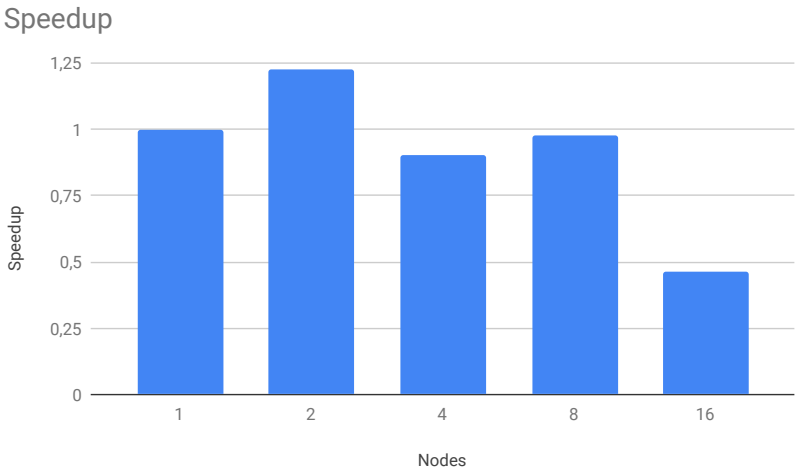


Figure 4.22: Speedup Bi-phase K-means weak scaling

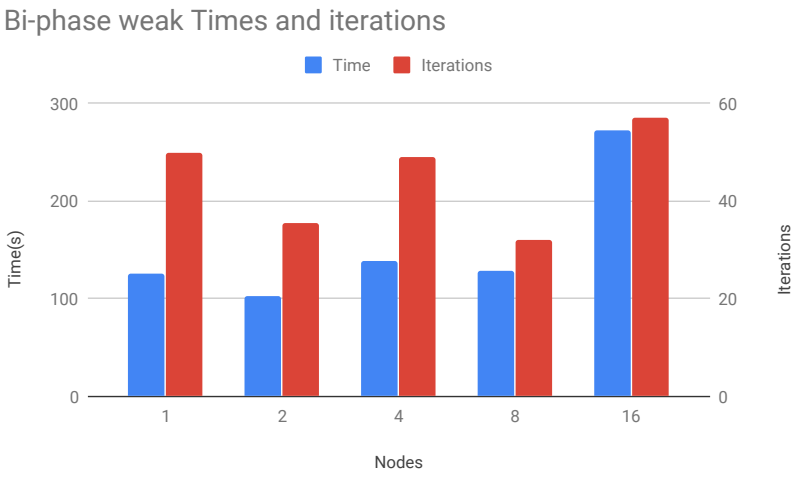


Figure 4.23: Scalability Loss Bi-phase K-means weak scalings

4.3.6.2 Multi-phases tests

For this optimization we also wanted to define a fixed value to define the number of phases and amount of percentage data to be sampled for each phase. Initially We compared the first two approaches that we had looked up previously and decided to choose the *Geometric* increase because it showed better results than *Linear* increase. Geometric approach increases the percentage data in powers of 2 starting from 10% until reaching all data. The results are shown in graph 4.24, with a data size of 26G and 8 nodes we compare the run-times and iterations of both approaches.

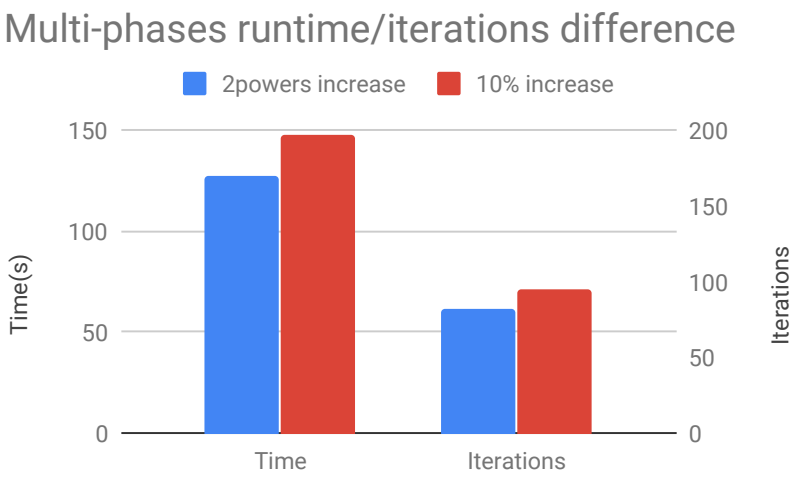


Figure 4.24: Multi-phases run-times/iterations difference, size=26G

A set of scaling tests for this optimization is shown in graph 4.25. The results also present an improvement compared to normal K-means, and still show bad scalability going further than 16 nodes.

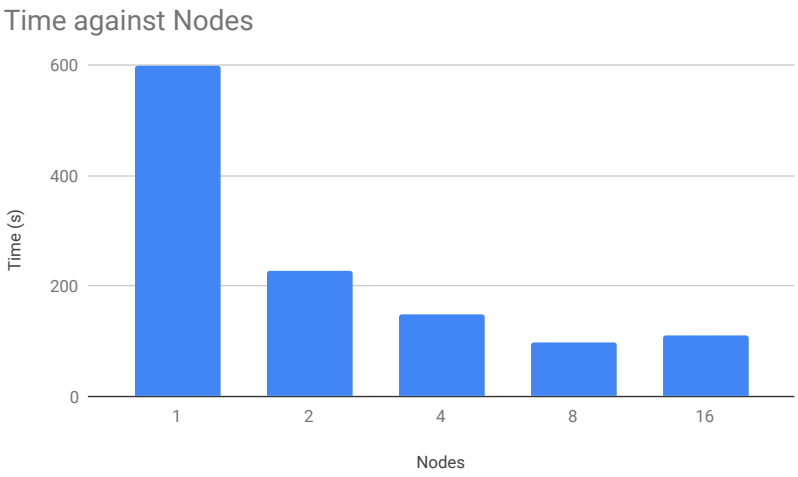


Figure 4.25: Multi-phases run-times over nodes, data size=26G

Moreover, the weak scaling test for Multi-phases optimization presents similar results than the Bi-phase weak scaling test. Similarly, for nodes {1,2,4,8,16} the data size is {6,4G,13G,26G,52G,105G}. We can see that for 105G data size the speedup decreases 4.26.

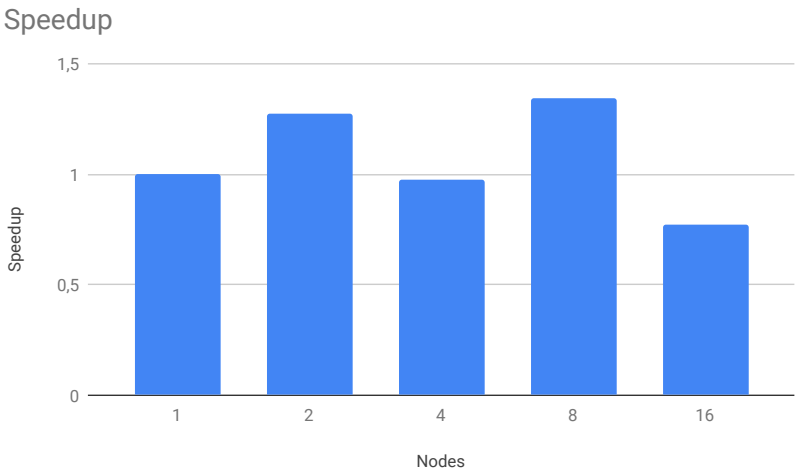


Figure 4.26: Speedup Multi-phases K-means weak scaling

Graph 4.27 displays the amount of time and amount of iterations spent in the same weak scaling test. The last execution with 16 nodes consumes more time as expected due to bad scalability.

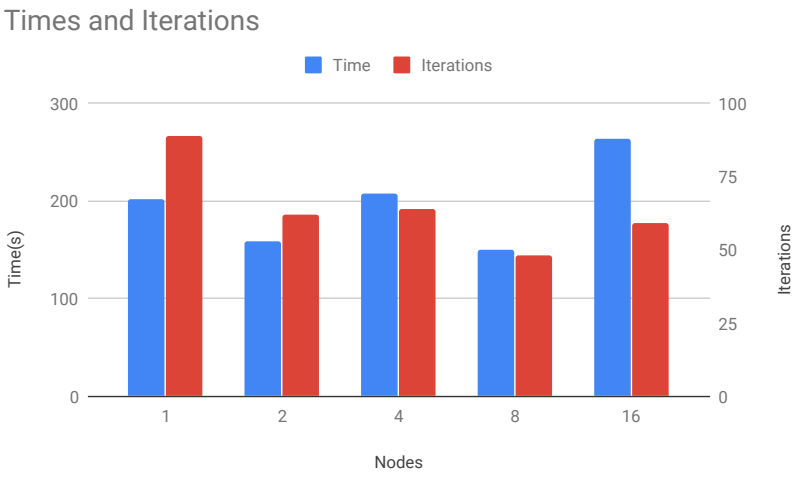


Figure 4.27: Times and iterations Multi-phases K-means weak scaling

Taking into account that our optimizations allow us to save energy consumption because they are faster than Normal K-means, another benefit that we have achieved for both Bi-phase and Multi-phases optimizations is that for the first phase, we obtain almost the same performance with few nodes than using many nodes 4.28. Because the amount of data is low, we don't have speedup when distributing the computation. Hence, in this phase (and possibly other phases for multi-phases depending on the amount of data) we can save resources by allocating fewer resources for a given phase, and then re-allocate the rest for the next phases. For instance, in most cases we will have the first phase in Bi-phase to be the most expensive phase in terms of time (>=50%). Performing a 16 node experiment would allow us to reduce to 2 nodes in the first phase, and thus reducing up to near 50% of the energy consumption.

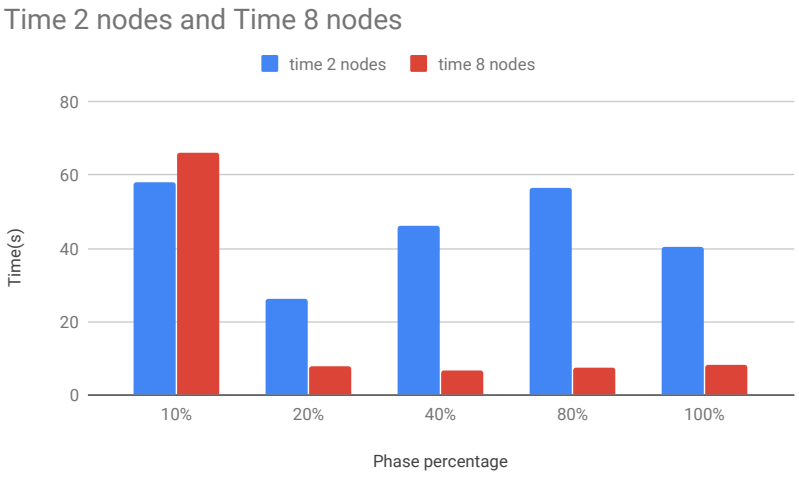


Figure 4.28: Multi-phases time comparison 2 nodes vs 8 nodes

4.3.7 Comparison between algorithms

After these algorithms parameters have been determined based on these preparatory experiments, a comparison of the run-times and number of iterations of the different implementations was made when clustering the small data set (26G). Table 4.5 shows results using different number of nodes for the three implementations, two approaches for Multi-phases. We tested the last approach for Multi-phases optimization (*Order of magnitude*) on a later instance than the previous scaling tests and found out that it performed better than *Geometric* when using few nodes, even though the previous scaling tests have been performed with *Geometric* approach, in the following table results for Multi-phases *Order of magnitude* are also added. The Bi-phase and Multi-phases *Order of magnitude* optimizations compete for the best performance. Bi-phase is up to 4 times better than Normal K-means, *Order of magnitude* gets very good performance with few nodes, surpassing Bi-phase until 4 nodes.

All implementations suffer from scalability issues, because in the end we are using the native K-means|| algorithm from MLlib, though handling the data differently.

Number of nodes	1	2	4	8	16
Normal K-means	1688,475	900,569	434,827	178,766	138,425
Bi-phase K-means	421,525	213,885	127,155	83,255	80,887
Multi-phases *Geometric* K-means	597,846	277,444	148,873	96,583	110,926
Multi-phases *Order of magnitude* K-means	210,973	152,307	135,242	129,001	105,473

Table 4.5: Run-times(in seconds) vs Number of nodes, data size=26G

Table 4.6 shows results about the number of iterations that are related with the times from 4.5. Normal K-means has the same number of iterations because it is a fixed problem and we are only increasing the amount of resources available. On the other hand we have selected the mean number of iterations for Bi-phase and Multi-phases, because the sampled data is not the same for each execution the number of iterations is not equal but very similar.

Number of nodes	1	2	4	8	16
Normal K-means	173	173	173	173	173
Bi-phase K-means	38,24	38,24	38,24	38,24	38,24
Multi-phases *Geometric* K-means	48,44	48,44	48,44	48,44	48,44
Multi-phases *Order of magnitude* K-means	28,21	28,21	28,21	28,21	28,21

Table 4.6: Iterations vs Number of nodes, data size=26G

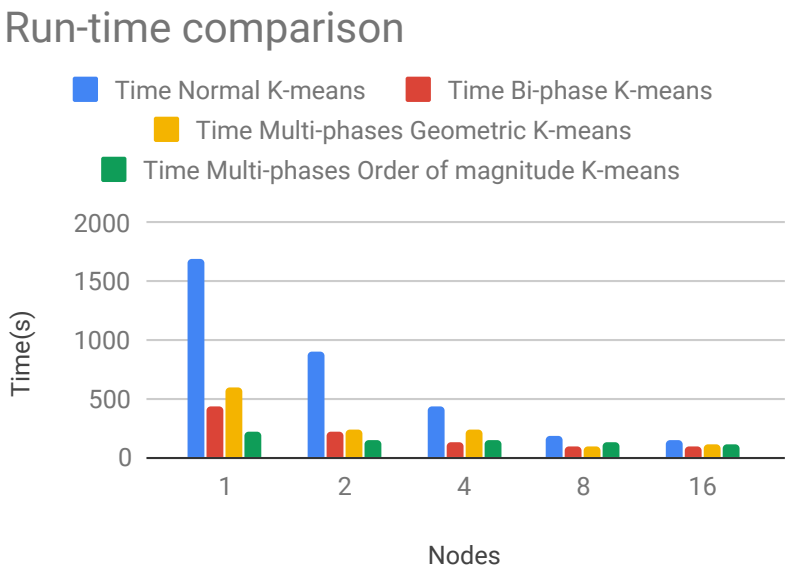


Figure 4.29: Run-time comparison over nodes, data size=26G

4.3.8 Efficient data sampling

We initially intended to perform similar tests with Qbeast, unfortunately the protected IP wasn't ready as we planned, we could only perform tests with small data sets (<1G) and that wouldn't follow the criteria from the previous tests. Therefore, we decided to focus on the validation of our theory.

We have previously mentioned the advantages that we could get using efficient sampling, in this section with the help of graphs to better visualize it, we will show the times to read data and K-means executions for each phase. We will use results from previous executions using Spark read to load the data from disk, and analyze how it would get improved with efficient sampling.

Spark sample function requires to read all data, doesn't matter the source, to then filter out the desired sample, which is inefficient when you only want to load a small percentage of data. Efficient sampling allows us to read small percentages of data without the requirement to read all.

These tests have been done in 8 nodes, and the same data set (26G) used like in most of the previous tests. Normal K-means read and K-means execution times are shown in Figure 4.30, it is only a single read of all data and a K-means execution since we are running the algorithm without any modification.

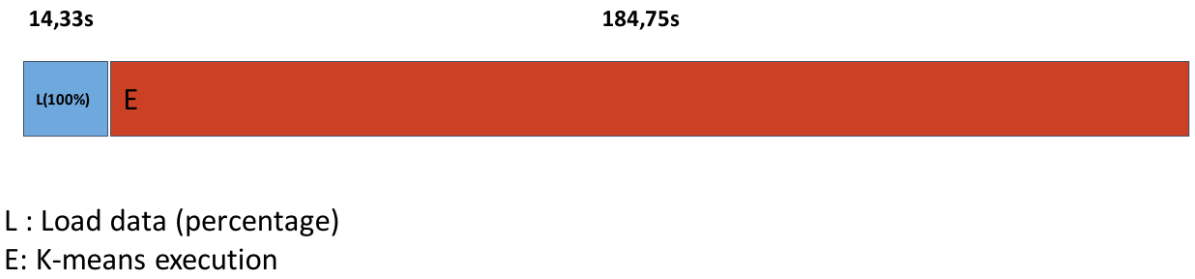


Figure 4.30: Normal K-means

Bi-phase read and execution times can be found at Figure 4.31, on top we have the times measured without efficient sampling (Spark sample function), on the bottom we have an approximation of the times if we were to use efficient sampling. Since we could query just a small percentage of the data without the need to compute the whole data set, we would be able to load the initial query much faster, followed by the first K-means execution phase whilst loading the rest of the data ideally before the last phase.

As we can observe, when using efficient sampling we reduce the read time relatively to the percentage data sampled. We go from 21,44s to load 1% of the data, which in reality is reading all the data to then filter out the 1%, to only spending approximately 1% of that time ~ 0,2s.

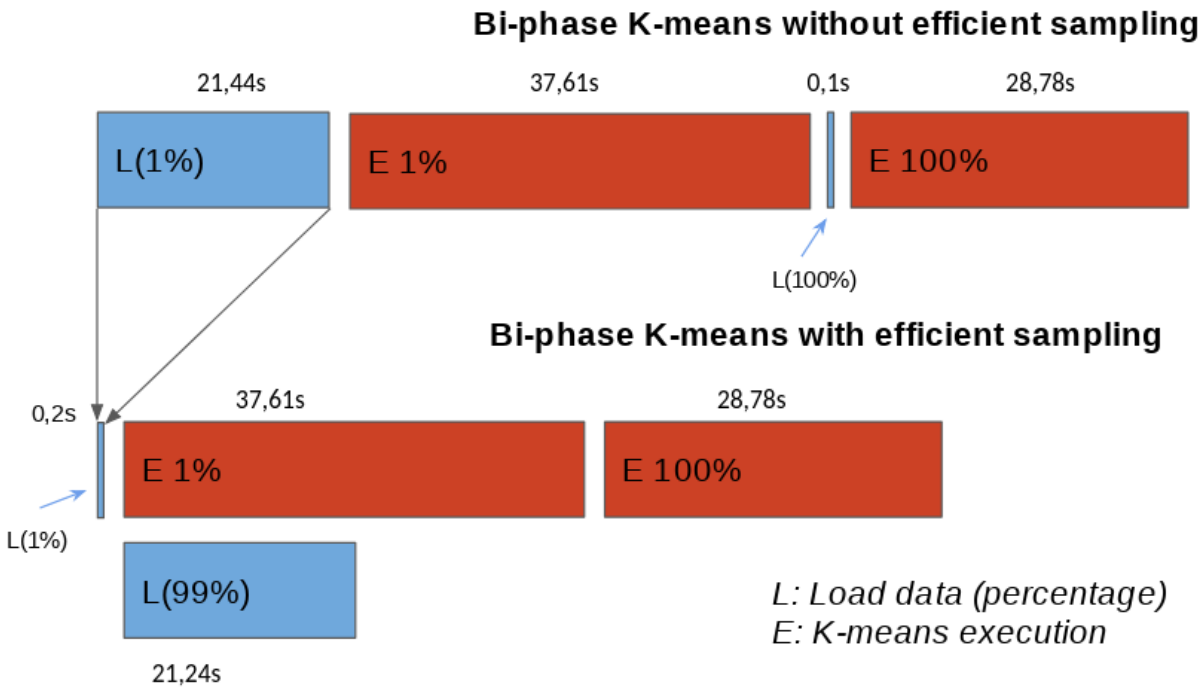


Figure 4.31: Bi-phase K-means

Likewise, Multi-phases would benefit when using efficient sampling similarly as Bi-phase, but with more phases. In Figure 4.32, we can see the difference between using efficient sampling or not. In this case, the algorithm has 4 phases: {0.1%,1%,10%,100%}. After loading the initial small sampling using efficient sampling considerably faster than without it, then we should be able to load the rest of the data while we execute the first

and second phase.

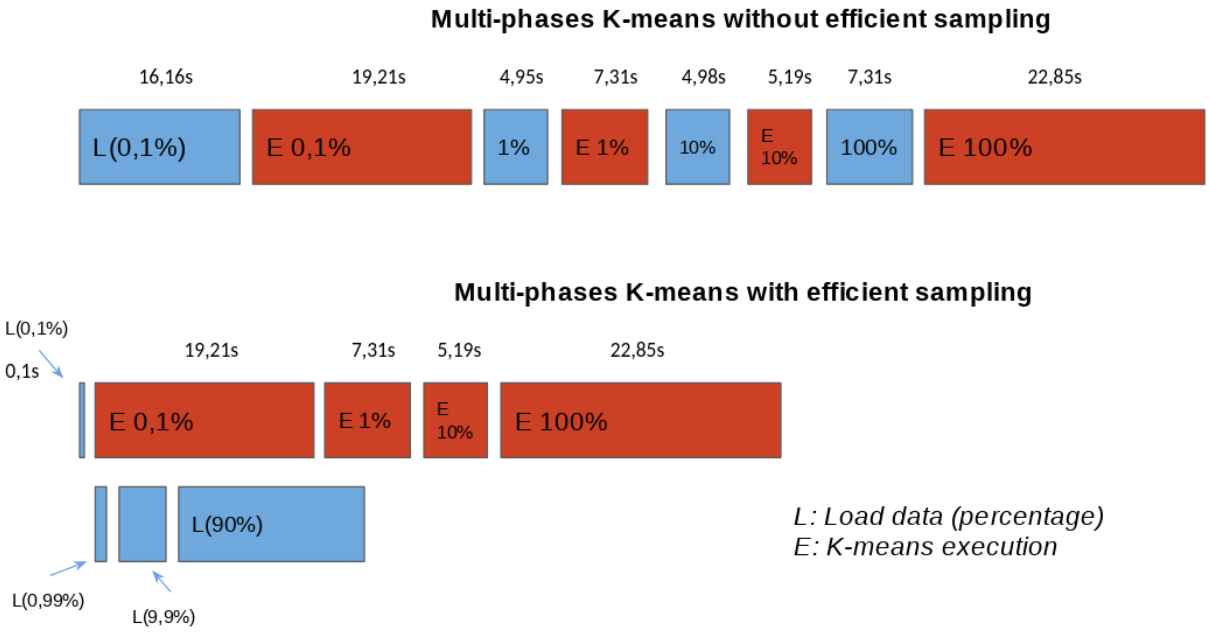


Figure 4.32: Multi-phases K-means

Finally, we have a comparison graph 4.33 where we can see the run-times of the algorithms when using normal sampling (Spark sample function) along with the approximated run-times when using efficient sampling (Qbeast sample) for Bi-phase and Multi-phases optimizations.

Figure 4.33: K-means comparison, efficient vs normal sampling

4.3.8.1 From Qbeast indexing to data storage

In this work, we have analyzed some of the possible advantages of using MIS with Machine Learning. In the current implementation, Qbeast offers indexing capabilities, which targets on retrieving a specific small subset of a large data set. With our work, we motivated the developing of a new storage architecture that offers MIS, and it is optimized for large batch operations, like the ones required in ML. For instance, we outlined a few possible improvements that are needed.

- **Better large query management**: Once we issue the query from Apache Spark, Qbeast starts to fetch all required data from the disk and temporarily stores the results in memory so that Spark can access it. Such a scheme works for small queries, but for the large ones, if Spark is not fast enough to consume the data, it can lead to memory issues. Thus we need to slow down Qbeast when the client cannot consume the data fast enough.
- **Native support for spatially stratified sampling**: When clustering data, some regions of data are dense enough that the remaining points may not need to be computed to know that they will form a cluster. Take for an instance a Catalonia map contained with data points from geo-tagged location tweets, once we start computing Barcelona tweets we know that they will form a cluster, the same would happen with Lleida or others. At some point we can decide to directly add to the Barcelona cluster the remaining points in the same region. Qbeast can provide the density of a region easily. Thus if we could avoid computing some regions of data because they are dense enough, we could improve performance.
- **Incremental sampling support**: Having the ability to sample percentages of data has shown very good results. Though, being able to read a specific data percentage after a sample has been performed would avoid unneeded computation. For instance, we have queried 1% of the data, when we query the 10% it would be better to only query the 9% left because we already have the initial 1%.
- **Columnar support**: Cassandra stores data in rows, as it is the best format for accessing to single elements, however, ML algorithms access to large groups of items, usually accessing each column independently. Therefore a different data format, a columnar one, may be analyzed to improve performance.

Chapter 5

Conclusions

The goal of the project was to study Machine Learning algorithms and work on implementations taking advantage of multidimensional indexing with efficient sampling. We studied several clustering algorithms and ended up selecting K-means for a more thorough study and work on optimizations.

One thing that we didn't expect at the start of the project was that setting up the Spark environment is not a trivial task, as well as studying the scalability of the algorithms, task partition and overall the good distributed computation when using Spark. These tasks took more time than we initially thought but were new and interesting to work on.

Furthermore, it has been shown that clustering algorithms suffer from bad scalability, and computing high amounts of data can become a bottleneck. Precisely, K-means can be benefited by our optimizations, while not losing any quality in the results, the experiments have showed that both Bi-phase and Multi-phases improve the normal execution of K-means in time and computational cost. The optimizations also present good results when reading the data from Spark read [36] but they have a lot of potential when using Multidimensional Indexing with efficient Sampling (MIS), since we can query small percentages of data without needing to compute the whole data set. Moreover, these optimizations allow us to save run-time, which also translates into resource saving.

Taking into account that our optimizations allow us to save energy consumption because they are faster than Normal K-means, another benefit that we have achieved for both Bi-phase and Multi-phases optimizations is that for the first phase, we obtain almost the same performance with few nodes than using many nodes. Because the amount of data is low, we don't have speedup when distributing the computation. Hence, in this phase (and possibly other phases for multi-phases depending on the amount of data) we can save resources by allocating fewer resources for a given phase, and then re-allocate the rest for the next phases. For instance, in most cases we will have the first phase in Bi-phase to be the most expensive phase in terms of time (>=50%). Performing a 16 node experiment would allow us to reduce to 2 nodes in the first phase, and thus reducing up to near 50% of the energy consumption.

In this work, we have analyzed some of the possible advantages of using MIS with Machine Learning. In the

current implementation, Qbeast offers indexing capabilities, which targets on retrieving a specific small subset of a large data set. With our work, we motivated the developing of a new storage architecture that offers MIS, and it is optimized for large batch operations, like the ones required in ML.

Lastly, as a personal conclusion we can say that this project has been fairly interesting because it provided me experience in a field that I considered attractive, as well as the satisfaction that these optimizations have a big potential to be used in real world applications. Also, it has been an enriching experience in both academic and professional level, as it is a field very studied in the actual market, and I have been able to learn and acquire notions from new tools.

5.1 FUTURE WORK

Overall we are happy with the work done in this project. Though, we know that further optimizations that we couldn't implement have potential, as well as other clustering algorithms apart from K-means could be optimized.

5.1.1 DBSCAN

As we have studied DBSCAN, we are aware that the scalable algorithm could be benefited by region queries using Qbeast. But for that, we needed to implement our own merge function for the algorithm because Qbeast manages data on its own way.

5.1.2 Resource re-allocation

When analyzing the tests for Bi-phase and Multi-phases, we realised that when querying smaller amounts of data and performing K-means we don't get a significant advantage for using more resources. Hence, we could de-allocate the number of nodes used to perform the initial phases where we have small amounts of data, and then re-allocate all the resources for next phases.

5.1.3 Dynamic-phases optimization

This optimization would aim to optimize K-means with a similar approach as Multi-phases optimization. The difference between the two is that for Dynamic-phases, it will force the end of a K-means execution, instead of waiting for the algorithm to converge by itself. A thorough study is required to analyze when would be optimal to force the convergence of an execution. The algorithm will obviously let it converge by itself in the last phase, but since it is performing multiple non-deterministic executions to just get the centers from each one, and the actual output for each phase is missing a considerable amount of data we don't need a "perfect" convergence in the first N-1 phases, where N is the total number of phases. This optimization would improve the total cost of the algorithm.

5.1.4 Asynchronous K-means

Another optimization that we had in mind, but could not fit in this project timeline was Asynchronous K-means. The idea would be to start a K-means execution with a small sample of data, start iterating the algorithm and increase in an asynchronous way the amount of data while iterating.

Bibliography

[1] Github sparklens. https://github.com/qubole/sparklens.

[2] Spark configuration docs. https://spark.apache.org/docs/latest/configuration.html.

[3] Spark resource allocation guide. http://site.clairvoyantsoft.com/understanding-resource-allocation-configurations-spark-application/.

[4] Spark standalone mode configuration docs. https://spark.apache.org/docs/latest/spark-standalone.html#cluster-launch-scripts.

[5] Spark tuning guide. https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[7] Apache. Apache licenses. http://www.apache.org/licenses/.

[8] M. Armbrust, A. Ghodsi, M. Zaharia, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, and M. J. Franklin. Spark SQL. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 1383–1394, 2015.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[10] D. Arthur and S. Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[11] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. *Proc. VLDB Endow.*, 5(7):622–633, Mar. 2012.

[12] D. Bajovic, D. Jakovetic, N. Krejic, and N. K. Jerinkic. Parallel stochastic line search methods with feedback for minimizing finite sums. In *19th European Conference on Mathematics for Industry*, page 132, 2018.

[13] H. B. Barlow. Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[15] C. Cugnasco, Y. Becerra, J. Torres, and E. Ayguadé. D8-tree: A de-normalized approach for multidimensional data analysis on key-value databases. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN '16, pages 18:1–18:10, New York, NY, USA, 2016. ACM.

[16] B. Y. J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53:72–77, 2010.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[18] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2Nd Edition)*. Wiley-Interscience, New York, NY, USA, 2000.

[19] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[20] C. Fraley and A. E. Raftery. How many clusters? which clustering method? answers via model-based cluster analysis. *The Computer Journal*, 41:578–588, 1998.

[21] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[22] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*, page 47, New York, New York, USA, 1984. ACM Press.

[23] D. Han, A. Agrawal, W.-k. Liao, and A. Choudhary. *A Fast DBSCAN Algorithm with Spark Implementation*, pages 173–192. Springer Singapore, Singapore, 2018.

[24] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[25] D. Jakovetić, D. Bajović, N. Krejić, and N. K. Jerinkić. Distributed gradient methods with variable number of working nodes. *IEEE Transactions on Signal Processing*, 64(15):4080–4095, 2016.

[26] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[27] A. Lakshman and P. Malik. Cassandra. *ACM SIGOPS Operating Systems Review*, 44(2):35, 4 2010.

[28] S. Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[29] M. R. Mahmud, M. A. Mamun, M. A. Hossain, and M. P. Uddin. Comparative analysis of k-means and bisecting k-means algorithms for brain tumor detection. In *2018 International Conference on Computer, Communication, Chemical, Material and Electronic Engineering (IC4ME2)*, pages 1–4, Feb 2018.

[30] O. Maimon and L. Rokach. *Data Mining and Knowledge Discovery Handbook*. Springer-Verlag, Berlin, Heidelberg, 2005.

[31] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[32] H. Neukirchen. Survey and performance evaluation of dbscan spatial clustering implementations for big data and high-performance computing paradigms. 2016.

[33] E. Schubert and A. Zimek. Elki: A large open-source library for data analysis - elki release 0.7.5 "heidelberg", 2019.

[34] J. G. Shanahan and L. Dai. Large scale distributed data science using apache spark. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 2323–2324, New York, NY, USA, 2015. ACM.

[35] Sparklens. Slides from sparklens. https://www.slideshare.net/slideshow/embed_code/key/pweFbIDCL2B2UW, 2018.

[36] Sparkread. Generic load/save spark functions. https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html, 2018.

[37] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. In *In KDD Workshop on Text Mining*, 2000.

[38] O. Wiki. Downloading data — openstreetmap wiki,. https://wiki.openstreetmap.org/w/index.php?title=Downloading_data&oldid=1631037, 2018. [Online; accessed 6-mayo-2019].

[39] Wikipedia contributors. K-means clustering — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=K-means_clustering&oldid=884530883, 2019. [Online; accessed 22-February-2019].

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[41] C. Zhao, X. Li, and Y. Cang. Bisecting k-means clustering based face recognition using block-based bag of words model. *Optik - International Journal for Light and Electron Optics*, 126(19):1761 – 1766, 2015.

[42] A. Zhou, S. Zhou, J. Cao, Y. Fan, and Y. Hu. Approaches for scaling dbscan algorithm to large spatial databases. *Journal of Computer Science and Technology*, 15(6):509–526, Nov 2000.